
# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# THESIS

**OBJECT TRACKING USING WIRELESS SENSOR NETWORKS**

by

Vlasios Salatas

September 2005

| | |
|---|---|
| Thesis Advisor: | Gurminder Singh |
| Thesis Advisor: | Arijit Das |

**Approved for public released; distribution is unlimited**

THIS PAGE INTENTIONALLY LEFT BLANK

| REPORT DOCUMENTATION PAGE | | | Form Approved OMB No. 0704-0188 |
|---|---|---|---|
| Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503. | | | |
| **1. AGENCY USE ONLY** ***(Leave blank)*** | **2. REPORT DATE**<br>September 2005 | **3. REPORT TYPE AND DATES COVERED**<br>Master's Thesis | |
| **4. TITLE AND SUBTITLE**: Object Tracking Using Wireless Sensor Networks | | | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S) Vlasios Salatas** | | | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | | | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)**<br>N/A | | | **10. SPONSORING/MONITORING AGENCY REPORT NUMBER** |
| **11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. | | | |
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT**<br>Approved for public released; distribution is unlimited | | | **12b. DISTRIBUTION CODE** |


**13. ABSTRACT (maximum 200 words)**

Wireless sensor network (WSN) is a promising new technology. It could be a way to achieve ubiquitous computing and embedded Internet. WSNs are an efficient solution for applications that involve deep monitoring of a deployment environment. The objective of this thesis is to explore the use of WSNs for object tracking and motion estimation. It introduces the WSN technology, their theoretical characteristics, system constraints, WSN architectures, deployment topologies and standards. The object-tracking system that this thesis introduces, demonstrates a real-world application that uses a WSN to track objects and communicate their information. It is an event-driven application implemented in Java, built on top of the Crossbow MSP 410 wireless sensor system. The algorithm process the data returned from the WSN detection signals and tracks the object's motion. Deployment scenarios are proposed that demonstrate suitable node topologies for the system. The evaluation of the object–tracking system is performed by conducting a number of indoor and outdoor experiments..


| **14. SUBJECT TERMS**<br>Wireless Sensor Network, Motion Detection, Object's Tracking, Node, Mote, Crossbow, MSP 410, Network Architecture, Nodes Topology, Active Message, | | | **15. NUMBER OF PAGES**<br>297<br>**16. PRICE CODE** |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT**<br>Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE**<br>Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT**<br>Unclassified | **20. LIMITATION OF ABSTRACT**<br>UL |

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public released; distribution is unlimited**

**OBJECT TRACKING USING WIRELESS SENSOR NETWORKS**

Vlasios Salatas
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**September 2005**

Author: Vlasios Salatas

Approved by: Gurminder Singh
Thesis Advisor

Arijit Das
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Wireless sensor network (WSN) is a promising new technology. It could be a way to achieve ubiquitous computing and embedded Internet. WSNs are an efficient solution for applications that involve deep monitoring of a deployment environment. The objective of this thesis is to explore the use of WSNs for object tracking and motion estimation. It introduces the WSN technology, their theoretical characteristics, system constraints, WSN architectures, deployment topologies and standards. The object-tracking system that this thesis introduces, demonstrates a real-world application that uses a WSN to track objects and communicate their information. It is an event-driven application implemented in Java, built on top of the Crossbow MSP 410 wireless sensor system. The algorithm process the data returned from the WSN detection signals and tracks the object's motion. Deployment scenarios are proposed that demonstrate suitable node topologies for the system. The evaluation of the object–tracking system is performed by conducting a number of indoor and outdoor experiments.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

# LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| AES | Advance Encryption Standard |
| AM | Active Messages |
| AP | Access Point |
| CAP | Contention Access Period |
| CCA | Clear Channel Assessment |
| CFP | Contention Free Period |
| COTS | Commercial-Of-The-Shelf |
| CSMA/CA | Carrier Sense Multiple Access with Collision Avoidance |
| CSV | Comma Delimited Text |
| DARPA | Defense Advanced Project Agency |
| ED | Energy Detection |
| FCS | Frame Check Sequence |
| FSK | Frequency Shift Keying |
| FTP | File Transfer Protocol |
| GTS | Guaranteed Time Slot |
| GUI | Graphical User Interface |
| IEEE | Institute of Electrical and Electronic Engineers |
| ISP | In System Processor |
| LLC | Logical Link Control |
| LQI | Link Quality Indication |
| LR-WPAN | Low Rate WPAN |
| LSB | Least Significant Bit |

| | |
|---|---|
| MAC | Medium Access Control |
| MFR | MAC Footer |
| MHR | MAC Header |
| MPDU | MAC Protocol Data Units |
| MSP | Mote Security Package |
| NTS | National Traffic System |
| OSI | Open Systems Interconnection |
| PHY | Physical Layer |
| PIR | Passive Infrared |
| PPDUs | Protocol Data Units |
| PRNET | Packet Radio Network |
| RTS/CTS | Request-To-Send/Clear-To-Send |
| SSCS | Service Specific Convergence Sublayer |
| SURAN | Survivable Radio Network |
| TSSR | Tactical Remote Sensor System |
| WLAN | Wireless Local Area Network |
| WSN | Wireless Sensor Networks |
| WPAN | Wireless Personal Area Networks |
| ZDO | ZigBee Device Objects |

# ACKNOWLEDGMENTS

This thesis is dedicated to my family. My lovely wife, Maria and our children Spyridon-Alex and Vasilios have constantly supported me with their love, encouragement and patient.

I would also like to acknowledge my advisors Gurminder Singh and Arijit Das for their continue guidance mentorship and support throughout this thesis. They have patiently encouraged my efforts with their inspirational comments. Without their help, this work would never reach the level of quality that I have always wanted in my life

THIS PAGE INTENTIONALLY LEFT BLANK

# I. INTRODUCTION

## A. BACKGROUND

Although the concept of wireless ad-hoc networks and infrastructureless communication has been researched for more than three decades, its subcategory, wireless sensor networks, is brand new. While still young, wireless sensor networks and their applications have attracted the interest of the research and industrial communities.

Wireless sensor networks are the realization of embedded networks, in which networks of interconnected devices are set into larger systems and environments. They behave as instruments capable of covering large areas providing detailed information. The sensor networks are a new aspect in the IT field and a great research area for the computer scientists, involving systems design, networking issues, programming models, distributed algorithms, data management, and security (Culler & Hong, 2004).

Sensor networks provide the ability to accurately observe and interact with systems and phenomena in “real time”. They aim to be part of a range of applications that includes environment observation, security enforcement, monitoring and management of machinery, and package and container protection. In addition, sensor networks may have an indoor use as control system for temperature and lights (Culler & Hong, 2004).

To work as an embedded network the sensor devices must be small, and preferably battery-powered, be capable of working for a long period of time, and be expendable as well. Moreover, because a sensor network can contain thousands of devices, they must be cost-effective to develop, deploy, program, use, and maintain. Thus, sensor networks could present a significant challenge for study and experimentation (Culler & Hong, 2004).

## B. OBJECTIVES

In the last two to three years a number of theoretical and/or simulation studies were done on the topic of object-tracking. While these studies are useful, they are too general and provide little guidance for the actual deployment of sensor networks for real-life location-tracking of an enemy.

This thesis focuses on developing an object-tracking application and prescribes sensor network configurations that work well with our algorithms. We implement our software using Crossbow hardware technology. The major issues addressed in this project are the evaluation and efficient use of a wireless sensor network product with no changes, in a real-world application, and efficient ways to algorithmically analyze the collected raw data from the specific wireless sensor networks product.

Although the focus is the development of a real-world application using wireless sensor networks, it also provides be a great opportunity to explore the new area of wireless communication overall.

## C. RESEARCH QUESTIONS

Our primary target is an exploration and study of the field of sensor networks. The study addresses the following questions.

What is a sensor network?

- What requirements do sensor networks address?
- What are their main ideas and concepts?
- What are the applications of sensor networks?

What are the associated technologies?

- What are the standards that sensor networks use?
- What are the existing hardware implementations that use this particular technology?
- What are the current software solutions that facilitate sensor networks implementations?

How can we build new applications using sensor networks systems?

- What do sensor network systems require for general or specific applications?

Are they applicable? And what is the appropriate design for sensor network implementations in order to combine them with other existing systems?

At this point we describe and evaluate the specific object-tracking application to be built.

## D. SCOPE

The study area of the thesis is an overview of the wireless sensor-network technology, an evaluation of a specific wireless sensor network product, and an implementation of an object-tracking application. Thus the study is divided into two parts. Part one is an overview of wireless sensor networks . it includes an evaluation of the product and an assessment of a specific sensor network system; general characteristics. The second part is based on the evaluation results; it describes the implementation of the object-tracking algorithm. The object-tracking application receives and uses the raw values returned by the sensor network system to produce clear and meaningful outputs. The outputs are then easily intergraded into a larger application, or are used independently as the output of a specific isolated application.

## E. METHODOLOGY

The object-tracking application is based on an existing Crossbow wireless sensor network product without making any additional internal changes. All the data evaluation filtering and algorithmic manipulation takes place in the base station (PC) where the network's data finally arrives. This choice supports the developer's intention to deploy an application by using a specific product with a minimum amount of changes. The application deployment includes the design and the implementation parts. The design stage is an iterative process and as the most critical stage requires theoretical knowledge of wireless sensor network technology. In addition, it requires sufficient knowledge of the abilities and specifications of the wireless sensor network product that it uses. The implementation part, although it may seem straightforward, has some programming difficulties. The Java program, which is our programming language choice, must implement precisely the algorithmic part of the design. Finally, the evaluation of the Crossbow product and the object-tracking application phases includes indoor and outdoor

experiments under different conditions and scenarios and by using different objects (human, car, etc.).

## F. THESIS ORGANIZATION

Chapter II provides an overview of the wireless technology, describes the concepts of wireless ad-hoc mesh networks, and introduces the new area of wireless sensor networks. It describes possible wireless sensor network applications and related architecture and networking issues, and also introduces the sensor networks' constraints and concerns. Finally, it apposes current related industry standards to this wireless technology.

Chapter III describes the software and hardware components related to the object-tracking application. It begins with Crossbow wireless sensor network products and continues with the software solutions that the company provides. The final part of the chapter contains a short description of the TSSRv3 system. This system is heavily used during the object-tracking application implementation as an interconnecting system which is fed with the object-tracking application's outputs.

Chapter IV describes the object-tracking application, including the application's design considerations and the configuration issues. The chapter then describes the algorithmic manipulation and the application outputs, based on the data returned from the Crossbow wireless sensor network. Finally, it describes the programming implementation of the above algorithm.

Chapter V includes our conclusions from the experimental results of the Crossbows products and the object-tracking application and makes deployment recommendations. Finally, chapter VI overview the entire study and makes recommendations for future research.

# II. WIRELESS SENSOR NETWORKS

## A. INTRODUCTION TO WIRELESS SENSOR NETWORKS

### 1. Development of Wireless Networks

The wireless communication field has had a long history; development in the field began years ago, at the beginning of the twentieth century. One of the first wireless paradigms was the War Department Radio Net established by the U.S. Army Signal Corps in 1921. By the end of 1933, this nationwide radiotelegraphic network could manage more than a million words annually. Interestingly, many of the early Morse wireless networks were the result of collaboration between the army and amateur civilian. Although many of the stations were operated by volunteer amateurs, the Army-Amateur Radio System (AARS), established in 1925, was a well organized system based within the Army structure (Callaway, 2004).

Though the early wireless networks were simple, compared to the today's, they used many of the ideas still in use today and demonstrated similar implementation difficulties. Most of the networks used the same tree-based structure as that of many modern wireless networks. And even before World War I the amateur operators used the ad-hoc network idea to overcome the transmission-range limitation when relaying their messages. The operator's availability restrictions, on the other hand, are an example of the early wireless networks weaknesses analogous to the power-consumptions and message-delay concerns of current systems. Medium Access Control (MAC) techniques also evolved, which enable networks to avoid the hidden-terminal problem, by using manual Request-To-Send/Clear-To-Send (RTS/CTS) messages. After World War II, the National Traffic System (NTS), as well, implemented standardized message format and an early multicast message option (Callaway, 2004).

Those early paradigms of wireless communication networks demonstrated the basic concepts and architectures of the later implementations. Limitations such as low transmission power, data throughput, and manual station operation indicated concepts similar to successor wireless ad-hoc mesh networks.

## 2. Ad-Hoc Networks: Introduction

While some of the concepts of ad-hoc networking are older, the first successful wireless data-communication network that applied the concepts of a random-access protocol and a medium access-control method was the ALOHA system. ALOHA is a packet-based, single-hop, two-radio-frequency channels project designed by the University of Hawaii to offer interactive data communication between the university and the Hawaiian islands (Callaway, 2004).

ALOHA's successor was the Packet Radio Network (PRNET) developed by the U.S. Defense Advanced Project Agency (DARPA) to improve the ALOHA project. The PRNET provides a multi-hop feature capable of supporting communication over a wide geographical area. The system's infrastructure includes mobile stations, mobile radio repeaters, and wireless terminals (Karapetsas, 2005). The PRNET introduced many new technologies and concepts for ad-hoc networks. First, it used direct-sequence spread-spectrum transmission, supporting up to 138 network entities with high data rates. Network management flow- and error-control were additional new elements also introduced by the PRNET system. As were routing-table update solutions, initiated to support point-to-point and broadcast routing (Callaway, 2004).

Although PRNET gave great actuation to the ad-hoc networks field, it was not without flaws. Limitations such as the relatively small network size (138 nodes), the nodes' size and power characteristics, and security issues were not addressed until the early 1980s with the development of the Survivable Radio Network (SURAN) (Callaway, 2004).

The decreasing cost of hardware and software, the heightening of computational power, and the capabilities of small mobile devices, combined with the Internet explosion in the early 1990s, triggered the research community's interest in ad-hoc networks. The systematic research in the ad-hoc area created a need for standardization in the protocols and technology. In 1990, the Institute of Electrical and Electronic Engineers (IEEE) established an 802.11 subcommittee for wireless local-area network (WLAN) standardization. The 802.11 standard, released in 1997, supports ad-hoc peer-to-peer,

infrastructure based, with access points (AP) connectivity. As its channel method the 802.11 standard uses carrier sense multiple access, with collision avoidance (CSMA/CA).

In the context of the current technology, the IEEE 802.11 wireless LAN is a common example, which many authors use to support the ad-hoc definition. Ad-hoc networking is a subset of infrastructure-less communication. Stallings (2002) defines the ad-hoc wireless LAN as "a peer-to-peer network (no centralized server) set up temporarily to meet some immediate need," and adds, "there is no infrastructure for an ad hoc network. Rather, a peer collection of stations within range of each other may dynamically configure themselves into a temporary network." Peterson and Davie (2003) define an ad-hoc mobile network as "a group of mobile nodes that form a network in the absence of any fixed nodes." Moreover, Kurose and Rose (2003) define the ad-hoc concept through the IEEE 802.11 as a system in which "stations can also group themselves to form an ad hoc network - a network with no central control and with no connections to the "outside world." Here, the network is formed "on the fly", simply because there happen to be mobile devices that have found themselves in proximity to each other, that have a need to communicate, and that find no preexisting network infrastructure (for example, a preexisting 802.11 BSS with an AP) in their location." Figure 1 demonstrates an ad-hoc network based on the above definitions.

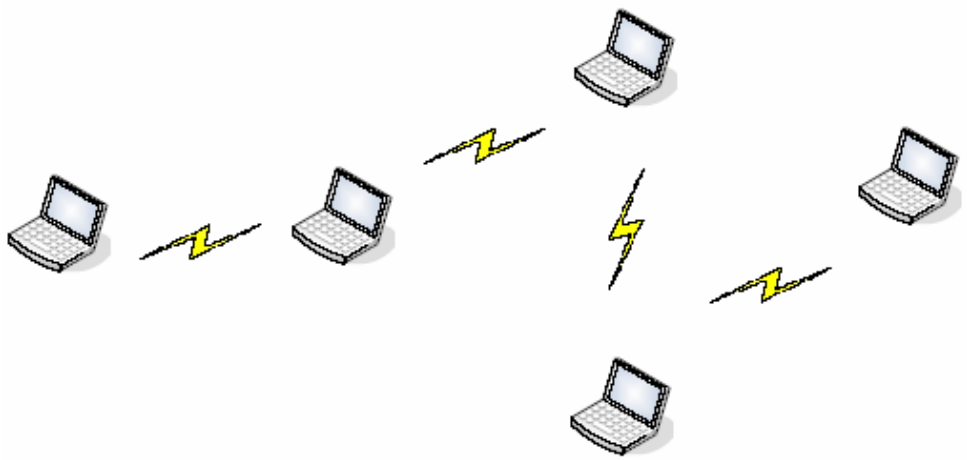


Figure 1. A simple ad-hoc network representation

## 3. Wireless Ad-Hoc Mesh Networks: Characteristics

A wireless ad-hoc mesh network can be considered as a subset in the ad-hoc networks category. As a subset, it inherits the valuable characteristics of the ad-hoc networks and optimizes their usage through a mesh topology. Thus, in general, a wireless ad-hoc network is defined by its ad-hoc capabilities and by the characteristics of the mesh topology that it uses.

The most important ad-hoc feature of a wireless ad-hoc mesh network is that it is self–organizing. This concept refers first to nodes' ability to identify their environment by discovering adjacent nodes. Wikipedia (Wireless Mesh Network, 2005) explains that, by using proper dynamic-routing protocols, the nodes update their routing tables and determine the most advantageous paths for forwarding the information hop-by-hop to its destination. The network formation is included in the system characteristics; it does not require any kind of coordination or administration; the network is formed "on the fly" (Kurose & Rose, 2004).

Feibel (1996) defines mesh topology as "a specific type of point-to-point connection in which there are at least two direct paths to every node….A more restrictive definition requires each node to be connected directly to every other node." The mesh network lacks a centralized base station; every node is free to communicate with any other network's node within its radio range. Figure 2 illustrates this layout.

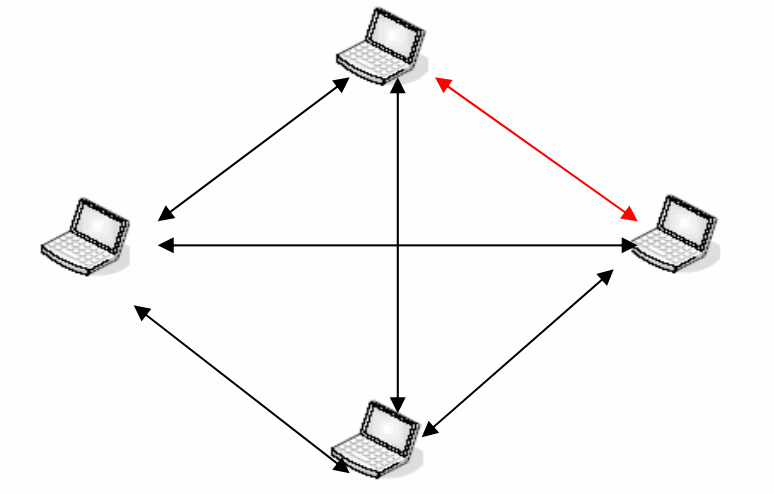


Figure 2. Mesh Topology

Mesh topology consolidates the above ad-hoc characteristics. It adds a self-healing feature, so the network, by supporting multiple connections for each node, is able to work efficiently even if it looses some of the nodes. If a system has a node failure, the node's neighbors find another route to forward their data. Thus the network is more reliable.

To summarize, wireless ad-hoc networks are self-organizing, self-healing, and adaptive. Ohrtman and Roeder (2003) refer to a wireless mesh networks as "an exciting new topology for creating low-cost, high-reliability wireless networks," and continues, "in a mesh network, each wireless node serves as both an AP and a wireless router, creating multiple pathways for the wireless signal." The redundant connections avoid the single-point-of-failure phenomenon and increase the system's robustness and reliability. Moreover, the wireless ad-hoc networks can easily be expanded, providing rapid area coverage. The mesh topology enables the related systems to support both fixed and mobile nodes. Finally, increasing nodes density provides more available bandwidth and increases the system's stability (Wikipedia "Wireless Mesh Network" webpage, 2005). Wireless ad-hoc mesh networks are a great step toward ubiquitous computing. The wireless sensor networks that the following sections introduce use the wireless ad-hoc mesh network concept, but their target seems to be mostly related to pervasive computing (Callaway, 2004).

## 4. Wireless Sensor Networks: Overview

The progress in wireless communications, digital electronics, and micro systems has enabled the development of small-size, low-cost, power-efficient multifunctional sensors. Moore's law predicts a great future for this technological field. In the future the typical sensor nodes the size "of a 35 mm film canister" (Wikipedia, Wireless Sensor Network Webpage, 2005), and their development cost will be drastically reduced, generating an explosion in the wireless sensor network usage.

Wireless sensor networks (WSN) is a rich domain that involves both hardware and system design. It consists of sensor devices that are "small in size and able to sense, process data, and communicate with each other, typically over an RF (radio frequency)

channel" (Haenggi, 2005). Their purpose is to collect and process data from the environment, produce a detection event and then forward the information to a specific destination.

Wireless sensor networks are a specialization of the wireless ad-hoc mesh networks. They inherit all the ad-hoc and mesh characteristics described above. They are wireless self-organizing, self-healing, and adaptive networks. They contain a large number of small, inexpensive, low-power nodes and use specialized communication techniques and routing, like "an asymmetric many-to-one data flow" (Carle & Simplot-Ryl, 2004) to communicate. Nodes' characteristics (size, lifetime, computational power), system's architecture, and protocols enable WSN to be deeply embedded into the environment. If these capabilities will be combined with the Internet, an "embedded Internet" (Culler & Hong, 2004) will be produced. Zhao and Guibas (2004) accent that "sensor networks extend the existing Internet deep into the physical environment. The resulting network is orders of magnitude more expansive and dynamic than the current TCP/IP network." Figure 3 provides an illustration of a sensor network and Internet integration. A complete WSN implementation is a "macroscopic view" (Carle & Simplot-Ryl, 2004) of the environment, implementing pervasive computing, it "enable us to observe and interact with physical phenomena in real time at a fidelity that was previously unobtainable." (Carle, & Simplot-Ryl, [2004]). WSN is a new, interesting, and active research area; it introduces various challenges and concerns; the following section highlights some of them.

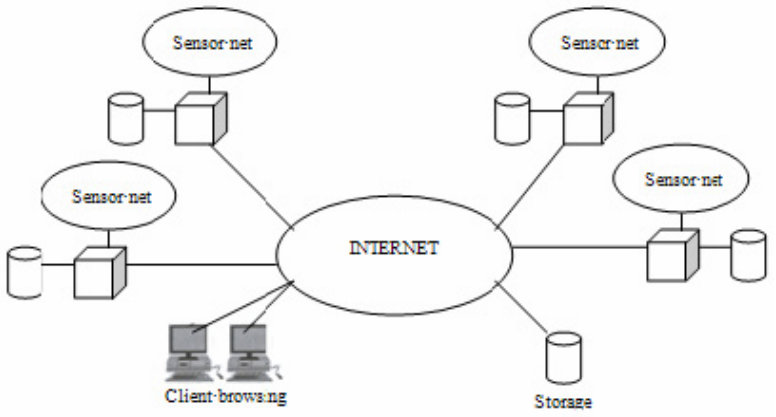



Figure 3. Integration of a Wireless Sensor Network and the Internet (Zhao & Guibas, 2004).

### 5. Wireless Sensor Network: Constraints and Challenges

Martin Haenggi (2004) specifies precisely the basic characteristics that the WSN have, including the following:

Self-organizing capabilities

Short-range broadcast communication and multi-hop routing

Dense deployment and cooperative effort of sensor nodes

Frequently changing topology due to fading and node failures

Limitations in energy, transmitted power, memory, and computing power.

They also highlights that the WSN differ from the wireless ad-hoc mesh networks in the latter three characteristics. Zhao and Guibas (2004) identify “limited hardware,” “limited support for networking,” and “limited support for software development” as general WSN design and implementation challenges. Wang, Hassanein, and Xu (2005) add “data redundancy,” the diversity of the possible application, and security and privacy concerns. Before some of the above concerns are analyzed further, we will discuss in the following section the important WSN applications that set the requirements and drove a WSN development.

## B. WIRELESS SENSOR NETWORKS: APPLICATIONS AND MOTIVATION

Implementations that use computers have existed for a long time, and computing is already an integral part of our life, heavily used in many aspects of our civilization. In addition to traditional systems, the sensor networks concept began to improve the computer’s applicability. Sensor networks’ pervasive attributes make them adaptable to a great range of applications. Their design supports high-level information-processing tasks such as “detection, tracking, or classification” (Zhao & Guibas, 2004). Culler, Estin, and Srivastava (2004) see a rough differentiation of sensor networks’ applications into three monitoring categories related to space, things, and the interaction of things with each other and their environment. The overview of possible applications for wireless sensor

networks that follows is based on the Haenggi (2005), Callaway (2004), Culeer, Estin, and Strivastava (2004), and Culler and Hong (2004).

### 1. Industrial Control and Monitoring

The deployment of wireless network sensors in the industrial control-and-monitoring field seems very prominent. Normally, a factory has a control room to monitor and control the state of the plant and the condition of the equipment. Specific critical values, like temperature or pressure, are collected from the plant or the equipment. The values describe the plant's or the equipment's condition, which is then forwarded to the control room where it is evaluated. Traditionally, industrial control and monitoring requires the deployment of a complex, expensive wired network. Sensor networks can replace the wired network, providing reliable data transfer and reducing the initial deployment and maintenance cost.

Lighting, ventilation and air-conditioning are other possible areas for wireless sensors. WSN provide the flexibility to support dynamic changes in the environment. This is also enhanced by the WSN programming feature, which offers secure and balanced services (e.g., balanced heating and air conditioning). When used to control and monitor complex equipment like robots, or other rotating and moving equipment, WSN provide the necessary flexibility. Thus, the system's reliability is increased, because damage caused by the machinery's movement is avoided. In addition, small-size sensing nodes can be used where wired implementations are impossible.

### 2. Home Applications

Home automation is another large application area for wireless sensor networks. The uses in the industrial applications field described above also apply to home implmentations. Centralized control of home appliances has already been implemented by using wired solutions or other wireless technology solutions. Their replacement by a wireless sensor network provides a development and maintenance cost reduction, system flexibility, and stretch ability. WSN also provides total, and secure control of the home devices. Another area for the use of WSN that is relevant to home application is the toy industry, a large market. The nature of wireless networks enable toys to behave in complex and logical ways at a reasonable cost..

### 3. Environmental and Agricultural Monitoring

Culeer, Estin, and Strivastava (2004) refer to the environmental monitoring of WSN implementations as pioneers in this technology. Wireless networks can be used for habitat monitoring and ecosystem measurements. Haenggi (2004) finds that seismic activity, forest fire, floods and water quality also can be detected and localized by the use of WSNs. Culler and Hong (2004) claim that the outdoor deployment, low power operation, fault tolerance, data quality, and networking characteristics of WSNs are ideal for environmental applications. Moreover, given those characteristics, WSNs can be used for agricultural purposes. Better knowledge of the agricultural environment enables the more precise control of fertilizers, water management cost reduction, quality maximization and environment protection.

### 4. Military and Security Applications

As with almost any new technology military and security application are recommended uses for wireless sensor networks. WSNs can assist or replace quards around a building or camp perimeter. Target localization and identification is another potential use, whereby friendly troops use WSNs to identify themselves (Callaway, 2004). Haenggi (2005) finds that such implementation can improve “military command, control, communication and computing (C4)” schema. Additionally, he describes an application for “surveillance and battle-space monitoring” in which the proper sensors are deployed in the ground or are carried by unmanned vehicles to monitor opposing forces. Haenggi (2005) mentions other potential uses in an “urban warfare” field: “to prevent reoccupation” of buildings that have already been cleared; and for “self-healing minefields,” where, instead of a “static complex obstacle,” the WSNs provide “an intelligent, dynamic obstacle that senses related positions and responds to an enemy breaching attempt by physical reorganization.”

### 5. Asset Tracking

Among the potential uses of wireless sensor networks, asset tracking is also a large area of interest for military and commercial application. Calllaway (2004) describes a possible use: for tracking “shipping containers both in a port and on a ship. By placing WSN nodes inside each container, it and its content become recognizable from a distance. An exact knowledge of the container’s type and position can save handlers a

great amount of time by preventing unnecessary errors. The WSNs provide a cost-effective way to increase the "shipper's productivity."

### 6. Heath Monitoring

Haenggi (2005) identifies two different wireless sensor network medical applications that are expected to rapidly increase. First, he mentions "medical sensing" in which data such as "body temperature, blood pressure, and pulse," collected from the system, can be transmitted to a local or remote computer for health monitoring uses. Additionally, WSNs can be used in the "micro-surgery" field, where tiny medical instruments are used to perform " microscopic and minimal invasive surgery."

### 7. Application Categories

The above applications show that, among the WSN applications, there are some common features. Holger and Willig (2005) identify the existence of data "sources" and "sinks" in most of the WSN applications in which the "sources" are the nodes that sense the data from the environment and the "sinks" are the nodes where the data arrived, like gateways. The "sinks" can be WSN components or they can sit outside the system. Holger and Willig (2005) place the applications based on the sources-and-sinks interaction in four categories. The first category is "event detection," is which the sources, when they detect an event send messages to the sinks. An event could be a single value, for example, an above threshold humidity, or a complicated type. Holger and Willig's second category is "periodic measurements," in which the sources periodically send messages to the sinks. The third category comprises "function approximation and edge detection" in which the WSN system, based on specific finite values, approximates an "unknown function." The final category is "tracking" in which the event producer is mobile, and thus a WSN is used to detect the object's position and possibly its speed and direction.

The preceding section included categories and possible implementations of wireless sensor networks. According to Haenggi (2005), the opportunities for the WSNs are "ubiquitous." Zhao and Guibas (2004) find that "the main long-term will be the increase in the number of sensors per application and the increase in the decentralization of sensor control and processing." However, the relevant constraints and challenges, that

are mentioned above will be further analyzed in the next sections. They must be addressed for easier and faster deployment of the wireless sensor network applications.

## C. POWER MANAGEMENT

Wang, Hassanein and Xu (2005) state that wireless sensor networks "outperform conventional sensor systems, which use large, expensive macrosensors to be placed and wired accurately to an end user." Despite the fact that they are revolutionary, affecting a great volume of applications, wireless networks have many constraints and challenges. One constraint perhaps the most important, is the system's limitation in its power supply lifetime. Most WSN system applications include a requirement for a maximum possible lifetime. In contrast, the core element of a sensor network is normally a battery powered node. As a result, the power management in wireless sensor network is extremely important.

Power management can be divided into two categories: the node's level and architecture and the topology system's power management. The next sections discuss those two approaches.

### 1. Node's Power Management

A wireless sensor network, in general contains four components. First, the microprocessor and memory unit is capable of performing the node's processing and logic tasks. Second, the sensor component is responsible for monitoring the environment. Third is the communication element which supports data transmission and reception. Finally, "a real-time micro-operating system controls and operates the sensing, computing, and communication units through microdevices drivers and decides which parts to turn off and on" (Wang, Hassanein, & Xu, 2005).

As Holger and Willig (2005) explain, the power management begin with the proper design and selection of the above components: "design low-power chips is the best starting point for an-energy-efficient sensor node." Holger and Willig also say that, in addition to the design optimization, careful control and operation of the nodes improves the energy efficiency. Normally, depending on the system's environment, the nodes do not detect great or frequent changes; thus a wireless sensor node does not have to operate continuously. The nodes' "dynamic power optimization" is described by

Wang, Hassanein, and Xu (2005) as a power management proposal. They note that the WSNs' workload is characterized by "burstiness." As a result, some parts of the nodes should switch to a lower power state between consecutive bursts. They identify the possible power states that a node can have as the following: "transmitting, receiving, ready, observing, standby, sleep, and off," (Figure 4). To maintain the system's functionality, QoS, and balance between power conservation and latency, the proper design and algorithms must be used. Additional power can be saved by varying the system's performance based on current needs. Wang, Hassanein, and Xu (2005) describe this variability as the "computational workload." They conclude that, currently, the workloads are "mostly nondeterministic" for producing an accurate model.

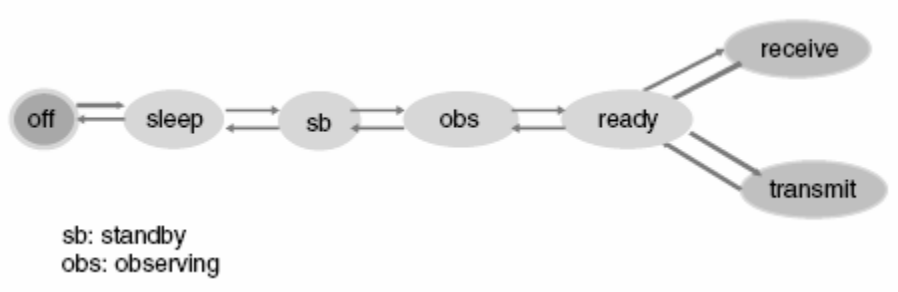



Figure 4. State transition diagram of a sensor node (Wang, Hassanein & Xu, [2005])

### 2. System's Power Management

Another alternative in the nodes' power management is transmission power optimization. It is included in the system's power management because a transmission power level adjustment affects many portions of a wireless sensor network system. The nodes' communication range, network topology and architecture, path selection, and retransmission rate are some of the aspects that are affected by transmission power. A power adjustment is restricted by propagation characteristics of the medium and by the nodes' limitations. The power level tuning can be made at the node level or at the system level (Wang, Hassanein, & Xu, 2005).

Application requirements and topology management also have an impact on the system's power management. An application that requires dense deployment for better area coverage and detection accuracy means that that node can reduce the transmission power, because the distance to the next one is small enough. This deployment produces

redundant data, however, that the system has to manage. In addition, the data must be transferred from the nodes to the end point for further manipulation. A proper selection of algorithms and communication protocols, like “rotate the node functionality periodically” or “traffic distribution and system partitioning” (Wang, Hassanein, & Xu, 2005), helps to maintain the energy balance among the nodes. For data processing to the nodes before its transmission, “data aggregation” (Wang, Hassanein, & Xu, 2005), or raw data forwarding is another choice that the system’s designer has to make, in trying to find the balance between latency and power consumption.

In summary, the power management in wireless sensor network systems is an important but difficult task. The designer has to compromise between the application requirements and the technological hardware restrictions. The nodes’ proper configuration, algorithm, and protocol selection, and the system’s correct architecture choice (presented in the next section) are required.

## D. TOPOLOGY ARCHITECTURE AND NETWORKING-ROUTING ISSUES

The development of wireless sensor networks is a new, rapidly growing technology that supports a great variety of applications. In the preceding sections we presented an overview of current and possible applications of WSN. In addition, we noted the constraints and challenges that the wireless network has to consider and overcome. Deployment strategies and systems architecture are closely related to the networking and routing issues. In wireless sensor networks, network characteristics and routing protocols illustrate the designer’s architectural intentions. The purpose of the following sections is to provide an overview of the design objectives and the possible deployment strategies and systems architectures.

### 1. Design Objectives

In wireless sensor network design the following design objectives aim to overcome the different challenges and to suffice the application based requirements (Holger & Willig, 2005, Wang, Hassanein, & Xu, 2005, Al-Karaki & Kamal, 2005)

#### *a. Sensor Devices*

Many wireless sensor applications require the use of a large number of sensing devices. Thus the sensors must be small so they don’t disorder the environment;

they must be cheap to reduce the application's total cost; and they must be compact so that they can be used outdoors and are energy efficient.

***b. Scalability, flexibility, and QoS***

To support a low deployment cost, easy maintenance, and expandability, the system must be scalable and flexible. In addition, the wireless sensor network's design should eliminate data redundancy by using in-network data aggregation, localized processing, and data fusion, which support efficient, accurate, and on-time data delivery. Finally, a system's QoS reliability and fault tolerance must be in balance with the resource constraints and application requirements. Careful design, proper systems architecture, and routing protocol selection will accommodate the above objectives.

***c. Application-Specific and Resource-Efficient Design***

Resource-efficient design-and-application requirements are critical. Most of the time, the system's architecture and protocols must be application specific; a universal design is not currently available. Additionally, power-saving techniques must be adopted to maximize the system's lifetime.

***d. Self-Configuration and Adaptability***

It is possible for a WSN's application to use a large number of nodes and, sometimes, to deploy them randomly. To correspond to the challenges, nodes have to be self-configurable and able to establish and maintain network connectivity. The network connectivity should be performed and maintained automatically, whenever nodes experience failures or change states.

***e. Locality of Information***

The exact position of a node is very important in wireless sensor networks. Only with knowledge of the nodes' location can the related data have meaning. It also makes network discovery, data and query addressing, and network maintenance easier.

***f. Attribute-Based Naming and Data Centric Routing***

In other network paradigms, data is requested from the sender. In wireless sensor networks, the data request is based on certain attributes, not on the node's address. And, in data-centric networks like WSNs, the nodes do not need to have a unique ID.

### *g. Cross-Layer Design*

Wireless ad-hoc networks, like all traditional networks, use a layered protocol stack. This approach has many advantages: simplicity, robustness, and scalability. However, each layer in the stack is isolated. The WSN's resource constraints are unable to support a traditional layered design. A cross-layer stack is probably the best solution, to support real-time data collection and transmission with limited resources. Figure 5 demonstrates a possible cross-layer protocol stack for WSNs.

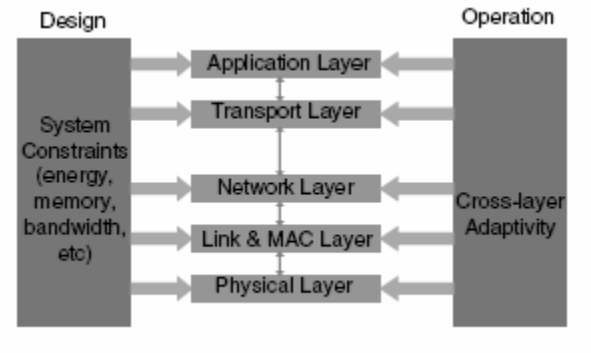



Figure 5. Cross Layer protocol stack for WSNs (Wang, Hassanein, & Xu 2005)

## 2. Topology and System's Architecture

Wireless sensor networks' characteristics and challenges, as well as the above mentioned design objectives, create a necessity for topology and systems architecture selection. The most restricted feature of WSNs, in relation to networking and routing, is the power constraint, because it reduces the available transmission power, which, in turn, affects the communication range of the individual nodes. Data transmission is the nodes' most consuming function. Wang, Hassanein, and Xu (2005) summarize this point precisely: "the energy consumed by communication is much higher than that for sensing and computation." Normally, the nodes are deployed in dense patterns to ensure coverage, communication channels, and detection precision.

Holger and Willig (2005) define the "topology control" that exists "to deliberately restrict the set of nodes that are considered neighbors of a given node. This can be done by controlling transmission power, by introducing hierarchies in the network and signaling out some nodes to take over certain coordination tasks, or by simple turning off some nodes for a certain time." Earlier, in the power management section, we introduced

ideas and proposals to minimize the network system's power consumption. This section will introduce recent research about wireless sensor networks' networking-routing architectures.

Al-Kraki and Kamal (2005), in their discussion about the WSN's routing protocols, note that, "in general, routing in WSNs can be divided into flat-based routing, hierarchical-based routing, and adaptive-based routing." Figure 6 summarizes the possible WSN architectures, some of which are introduced in sections below.

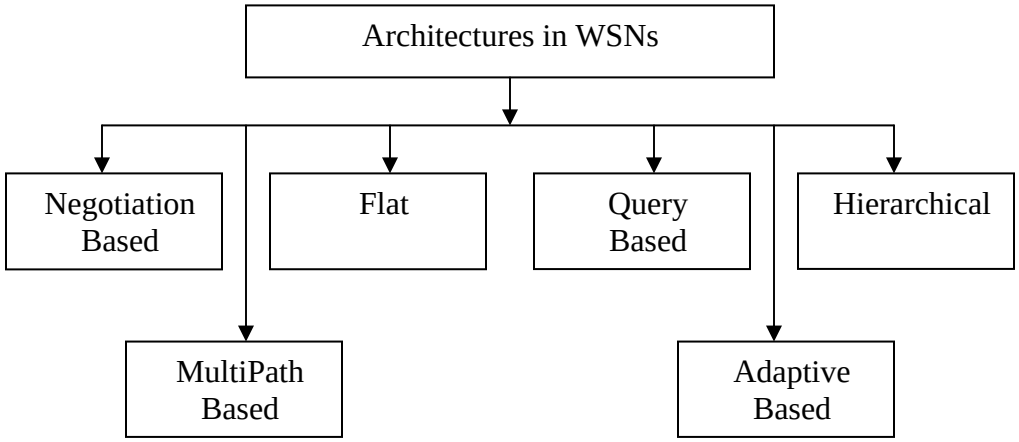



Figure 6. WSNs' architectures: An overview based on Al-Kraki and Kamal ( 2005)

***a. Flat Network Architecture***

In flat network architecture all nodes have identical characteristics and can perform the same tasks. Yarvis and Ye (2005) refer to the network's nodes as "complete interchangeable." In flat architecture, all the nodes are considered neighbors, and all are able to detect and forward data to the sink. Typically, by performing transmission power control and proper modulation, the number of neighbors can be reduced (Holger & Karl, 2005). Although that type of restriction improves network performance, Holger and Karl are against the notion of flat networks, because they produce differences between the nodes (heterogeneity). Various flat-architecture-based protocols have been proposed in the literature. Al-Kraki and Kamal (2005), for example, mention some of them: Sequence Assigment Routing (SAR), Directed Diddusio, and Minimum Cost Forwarding

Algorithm. Figures 7 and 8 present a dense flat network with and without transmission power control.

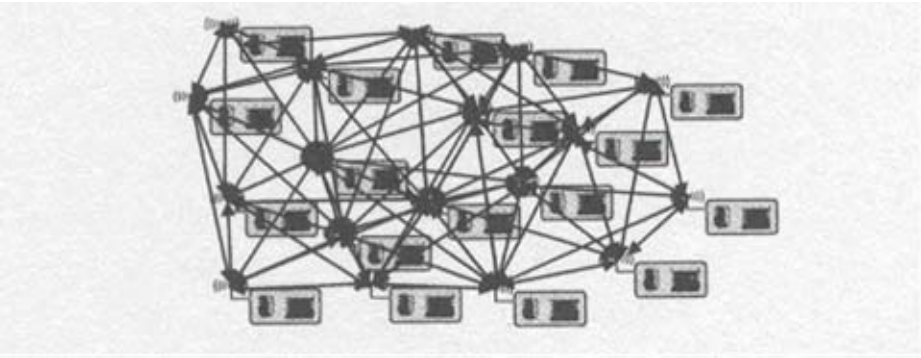


Figure 7. Topology of a dense wireless sensor network (Holger & Willig, [2005])

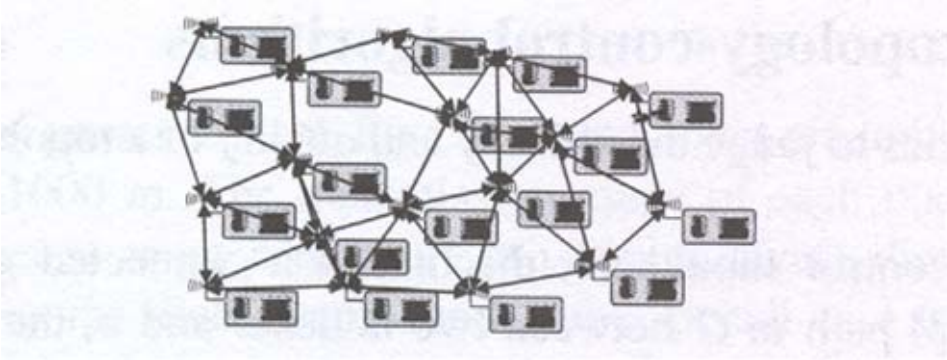


Figure 8. Wireless sensor network topology after reducing transmission power (Holger & Willig, 2005)

### *b. Hierarchical and Cluster-Based Network Architecture*

Hierarchical and cluster network architectures are not new topologies introduced in wireless sensor networks. They were originally used in wired networks. However, their "scalability and efficient communication" (Al-Kraki & Kamal, 2005) advantages are utilized by the WSNs, providing power consumption reduction. The hierarchical architecture assumes that the nodes are heterogeneous. Yarvis and Ye (2005) explain that in a tiered architecture "the functions of sensing, computing, and data delivery are divided unequally among nodes." The nodes belonging to the same level of hierarchy have the same tasks. Yarvis and Ye also note that the "functional decomposition of a sensor network can reflect physical characteristics of nodes, or it can simply be a logical distinction." Examples of roles for a node are sensing, data

aggregation, and backbone communication. If the nodes have the same characteristics, they can periodically change roles. Figure 9 demonstrates a possible three-tier architecture of a WSN, that is also connected to the Internet.

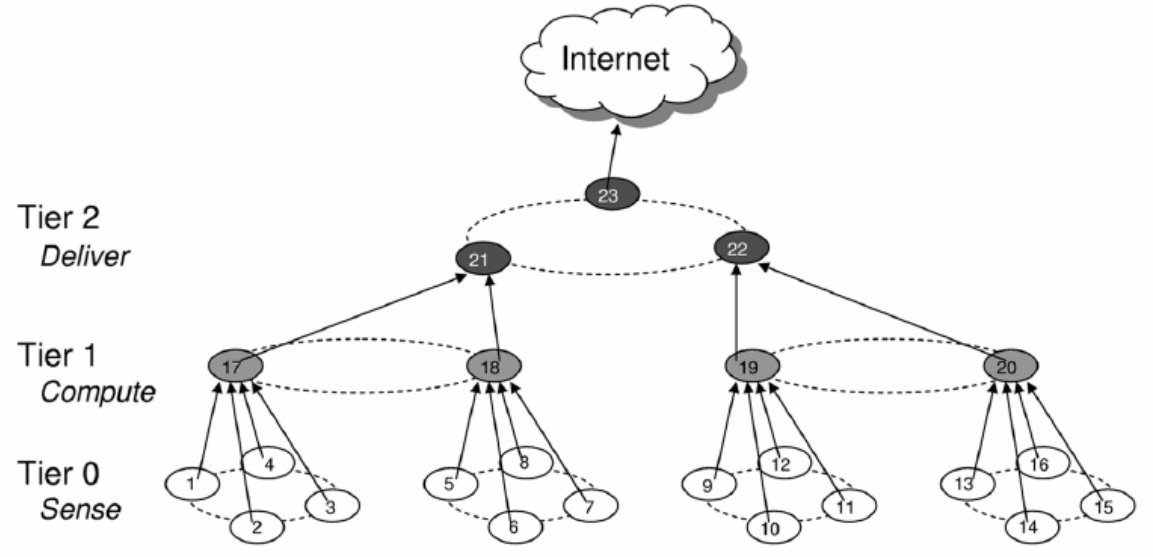



Figure 9. Example of WSN three-tier architecture (Yarvis & Ye, 2005).

According to Holger and Willig (2005), clusters are a "slightly different" architecture than the hierarchical; they are "subsets of nodes that together include all nodes of the original graph such that, for each cluster, certain conditions hold." Each cluster has a cluster-head; the rest of the nodes are a one-hop distance from the cluster-head. Wang, Hassanein, and Xu (2005) add that "clusters replace the one-hop long-distance transmission by multihop short-distance data forwarding." The cluster architecture characteristics allow simpler routing protocols to be used inside the cluster Guibas and Zhao (2004). Al-Kraki and Kamal (2005) mention some of the proposed hierarchical or cluster-based protocols: LEACH protocol, PEGASIS, TEEN and APTEEN, SMECN etc. Figure 10 exhibits a multihop clustering architecture.

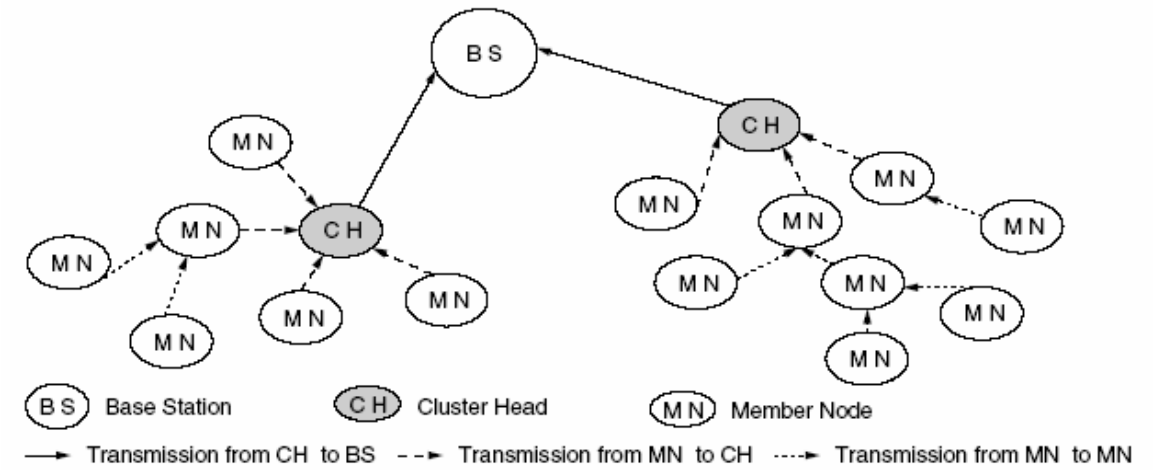



Figure 10. Multihop clustering architecture (Yarvis & Ye, 2005).

The three main network architectures described above, despite their advantages, also have drawbacks. The flat topology, although attractive, is very difficult to implement. On the other hand, the tiered architectures introduce hot spots and overhead into the network.

The three major architectures support a variety of routing protocols. Al-Kraki and Kamal (2005) introduce the adaptive routing, multiple routing, query-based routing, and negotiation-based protocols. A proper routing-protocol selection is based on the underline architecture and the application's specific characteristics. Because this section focuses on introducing the wireless-sensor-network field, it skips descriptions of routing protocols, and continues with an overview of different deployment strategies.

### 3. Deployment Strategies

Deployment strategies are an important aspect of wireless sensor networks. Although the sensors' deployments are application specific, they share a common objective. Different strategies aim to properly cover the area of interest by using the minimum number of nodes. Wang, Hassanein, and Xu (2005) place the deployment strategies in four different categories: "predetermined, self-regulated, randomly undetermined, or biased distribution." The following sections introduce the aforementioned categories.

### *a. Predetermined*

A predetermined strategy is used in a situation in which the deployment environment is known or a grid-based topology can be maintained. This strategy provides the ability to control the area coverage and the deployment cost maintaining a high QoS . There are two main difficulties of this strategy: first, that the deployment environment knowledge is a rare situation. Secondly, as the number of nodes increases the "computational complexity" also increases.

### *b. Self-Regulated*

The self-regulated strategy is the strategy that describes a scenario in which the nodes are deployed automatically in an unknown environment. Although a proper selection of the number of sensors maintains the deployment cost at an acceptable level, like the predetermined strategy, it has high computational complexity.

### *c. Randomly Undetermined*

The randomly undetermined strategy is the strategy suitable for the deployment of a large number of nodes in a hostile environment in which the nodes are not placed, but spread. The advantage of this strategy is that the deployment cost is low, but it cannot guarantee uniform coverage.

### *d. Biased Distribution*

Finally the biased distribution strategy can be described as a version of the randomly undetermined strategies. The nodes in this strategy are mainly deployed in a random manner, but, in specific geographical locations, the random deployment is biased.

## E. SECURITY AND PRIVACY CONCERNS

Holger and Willig (2005) underline that security, especially "network security is one of the most pressing concerns in all wireless networks, including wireless sensor networks." Privacy and security are crucial parts of wireless network system architecture. In addition, security and privacy are important requirements for many applications. The primary force for the development of security functions in WSNs is the military application area. Commercial applications also require security, but they are more interesting in the privacy issues. This section will present an overview of the security and privacy aspects of WSNs.

Slijepcevic, Wong, and Potkinjak (2005) note that WSNs have four main security-related properties, the first of which is the application's requirements and architecture. Wireless sensor networks provide enough flexibility to the designer to prioritize, adjust, and improve security and privacy aspects based on the application's requirements. To satisfy them, the designer has also to deal with the WSN's limited resources. The restrictions in energy, computational power, storage, and size constrain the possible security solutions. Slijepcevic, Potkinjak, and Wong (2005) state that there is a "trade-off between resources spent on security and the achieved protection." In addition to the resource constraints, the environment, in general, is hostile. The WSNs can be deployed in a battlefield or inside a forest. Moreover, the nodes can be visible and accessible to anyone. Finally, the nodes, in order to save power, prefer to perform additional computations to reduce the number of transitions in "in-network processing." This property is suitable for security implementations. However, in a situation in which a node may be captured, the adversary will have access to the security material.

Perrig, Stankovic, and Wagner (2004) point out that, because of the above properties and concerns, "traditional security techniques used in traditional networks cannot be applied directly." They also say that security is related to every aspect of the system design. The following sections discuss some WSN parts that are described in Perrig, Stankovic, and Wagner (2004).

### 1. Key Establishment and Trust Setup

WSNs properties, especially their node-limited computational capabilities, do not allow the use of traditional "key-establishment" solutions such as public-key cryptography. Moreover, the key-establishment becomes more complicated because of the system's scale and the communication patterns between the nodes. The shared-key solution also does not work because a node compromise allows decryption of the entire traffic. On the other hand, the solution that uses a symmetric key between each pair of nodes addresses the above problem, but it does not scale well. Another option is the use of a unique key between each node and the base station, but this makes the base a station single point of failure. Key distribution is an active research area. A recent proposal is a "random-key redistribution protocol," in which a pair of nodes uses a share key from a

pool. Different pairs of nodes must use different keys. In this solution, the adversary has to compromise a number of nodes to reconstruct the key pool. Further research and development in the last approach is expected.

### 2. Secrecy and Authentication

The common approach used to achieve secrecy and authentication is cryptography. Earlier WSN solutions involved link-layer cryptography, which is simple enough. To improve the security performance, later approaches propose "software-only cryptography." The University of California, Berkeley, implementation of TinySec is an example that improves the system's security with only 5%-10% performance overhead.

### 3. Privacy

Privacy issues have arisen based on the ubiquitous nature of WSNs, especially for commercial systems. The nodes' size, which become smaller, and the improvement of nodes' capabilities may support improper uses of WSN systems.

### 4. Communication Robustness

A denial-of-service attack is always possible in a WSN implementation, especially because of the low node transmission power. Currently, the spread-spectrum communication technique is the first measurement. Additionally, the networking characteristics of the WSNs can be used to avoid that kind of attack by rerouting the traffic through the system's unaffected parts.

To summarize, security is always a concern, especially in wireless implementations. As WSNs are a more restricted wireless communication, any kind of security-feature implementation seems more difficult. The standardization of wireless sensor networks that is introduced in the next section, by focusing the research community, could be a step toward to efficient solutions.

## F. PROTOCOLS AND INDUSTRY'S STANDARDS FOR WIRELESS SENSOR NETWORKS

Although many applications for wireless sensor networks are proposed, the communication protocols supporting them remain mainly unexplored and diverged. This is because the area of interest is new, each implementation is application specific, and the components of WSNs have a lot of constraints.

Wireless sensor networks use the wireless medium to communicate. However, the traditional communication protocols that support wireless communications, especially ad-hoc mesh networks, may not be well suited for them. WSN communication is an active research area, and many algorithms and protocols have already been proposed. This section introduces two proposed standards: the IEEE 802.15.4 and the ZigBee. The IEEE 802.15.4 covers the physical layer and the Medium Access Control (MAC) layer of low-rate Wireless Personal Area Networks (WPAN). The ZigBee is "an emerging standard that is based on the IEEE 802.15.4 and adds network construction (star networks, peer-to-peer/mesh networks, cluster-tree networks), application services, and more" (Holger & Willig, 2005).

### 1. 802.15.4

Holger and Willig (2005) mention home automation, home networking, and home security as possible applications for IEEE 802.15.4. They add that "most of these applications require only low-to-medium bitrates (up to some few hundreds of kbps), moderate average delays without too stringent delay guarantees, and for certain nodes it is highly desirable to reduce the energy consumption to a minimum."

The IEEE 802.15.4 standard (2003) defines the device types that can be used in a Low Rate WPAN (LR-WPAN). A device can be a Full-Function Device (FFD) or a Reduced-Function Device (RFD). The RFD can be used in simple applications in which they do not need to transmit large amounts of data and they have to communicate only with a specific FFD. The FFD can work as a Personal Area Network (PAN) coordinator, as a coordinator, or as a simple device. It can communicate with either another FFD or a RFD.

In keeping with the application requirements, the LR-WPAN operates in a star or peer-to-peer topology (Figure 11). In the star topology the RFD communicates with a single controller, the PAN coordinator. The PAN coordinator can perform the same function as the RFD, but it is also responsible for controlling the PAN; "it initiates, terminates, or routes communication around the network" (IEEE 802.15.4 Standard, 2003). The peer-to-peer topology supports ad-hoc mesh multihop networking. Any device in the peer-to-peer topology can communicate with any other device within its

communication range; however, this topology also has a PAN coordinator. All the devices in a LR-WPAN have a unique 64-bit address. This or a short address, allocated by the PAN coordinator, can be used inside a PAN. Additionally, each PAN has a unique identifier. The combination of the PAN identifier and the sort addresses allows communication across different PANs (IEEE 802.15.4 Standard, 2003).

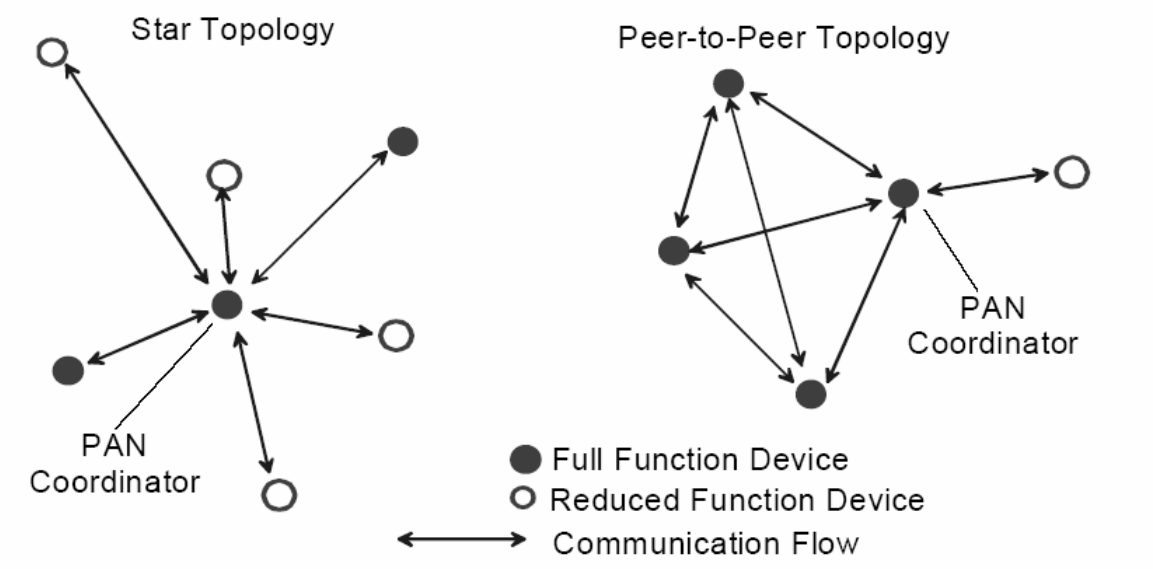



Figure 11. Star and peer-to-peer topologies in LR-WPAN: (IEEE 802.15.4 Standard (IEEE, 2003).

The LR-WPAN based on the open systems interconnection (OSI) seven-layer model has the layered architecture presented in Figure 12. The application and the network layer are the upper layers in the LR-WPAN architecture, but are outside the scope of the IEEE 802.15.4 standard. Only the physical layer, "which contains the Radio Frequency (RF) transceiver along with its low-level control mechanism," and the MAC layer "that provides access to the physical channel" are included in the standard and will be introduced in the following sections. Finally, the Type I 802.2 Logical Link Control (LLC) and the Service Specific Convergence Sublayer (SSCS) are intermediate sublayers supporting communication with the above layers (IEEE 802.15.4 Standard 2003).

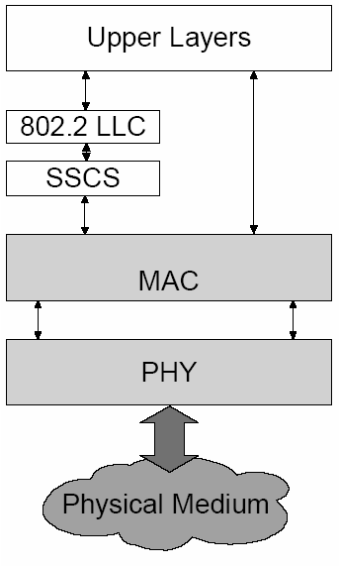



Figure 12. LR-WPAN architecture: (IEEE 802.15.4 Standard, 2003).

### *a. Physical Layer*

The 802.15.4 standard specifies two different services that the physical layer (PHY) provides. The PHY data service controls the radio, and thus, the transmission and reception of the PHY Protocol Data Units (PPDUs). In addition, the management service performs Energy Detection (ED) in the channel. The management service also performs Clear Channel Assessment (CCA) before sending the messages and provides Link Quality Indication (LQI) for the received packets.

Three different bands are defined by the standard. The 868-868.6 MHz for Europe, the 902-928 MHz for North America, and the 2400-2483.5 MHz worldwide. All of them belong in the Industrial, Scientific, and Medical (ISM) radio bands and they are using Direct-Sequence Spread Spectrum technique (DSSS). Each of the bands supports different a data rate and it uses a different modulation technique, chip rate, and number of channels. Moreover, if the system does not use the 2450 MHz frequency, it operates in both the 802 MHz and 902 MHz frequencies. The following table summarizes the supported bands and data rates (IEEE 802.15.4 standard, 2003).

| PHY (MHz) | Frequency band (MHz) | Modulation | Bit Rate | Number of Channels |
|---|---|---|---|---|
| 868/915 | 868-868.6 | BPSK | 20 | 1 |
| | 902-928 | BPSK | 40 | 10 |
| 2450 | 2400-2483.5 | O-QPSK | 250 | 16 |

Table 1. Frequency bands and data rates for IEEE 802.15.4 based on the IEEE 802.15.4 standard (2003).

The physical layer uses PPDU packets to communicate. Figure 13 demonstrates its structure. The least significant bit (LSB) is always transmitted and received first. The synchronization header (SHR), contains the preamble and the Start of Frame (SFD) fields, which helps receiver synchronization. The 8-bytes preamble contains only zero and is used for synchronization. The SFD contains a specific sequence of one and zeros and specifies the beginning of the frame. The PHY Header (PHR) contains the payload length. Packets with a length of 9 or more bytes are MAC Protocol Data Units (MPDU), as the next section further explains. Packets with length 5 are used for MPDU acknowledgements. The payload part of the PPDU encloses the MAC layer packet. Finally, the PPDU size can be up to 127 bytes (IEEE 802.15.4 standard, 2003).

| 4 bytes | 1 byte | 1 byte | | variable |
|---|---|---|---|---|
| Preamble | SFD | Frame length (7 bits) | Reserved (1 bit) | Payload (PSDU ) |
| SHR | | PHR | | PHY payload |

Figure 13. PPDU format based on the IEEE 802.15.4 standard (IEEE, 2003).

***b. MAC Layer***

The MAC layer is the interface between the SSCS and the PHY layer. Similar to the PHY layer, the MAC layer supports two services. The MAC data service is responsible for the transmission and reception of the MPDUs through the PHY data

service. The MAC management service, if the device is a coordinator, manages the network beacons. It is also responsible for PAN association and disassociation, frame validation, and acknowledgment providing "a reliable link between two peer MAC entities." In addition, it uses the CSMA-CA for channel access and handles and maintains the Guaranteed Time Slot (GTS) mechanism. Finally, it supports device security (IEEE 802.15.4 standard, 2003).

The IEEE 802.15.4 standard defines four different frame types: the beacon, data, acknowledgment, and MAC command frame. All frame types are based on the general MAC frame format (Figure 14). The frame control field describes and specifies the above different frame types. Every MAC frame comprises a MAC Header (MHR), which consists of a frame control, sequence number, and the information field. It also contains the MAC payload; different frame types have different MAC payload fields. The acknowledgment type does not have a payload. Finally, each frame includes a MAC Footer (MFR), which contains a Frame Check Sequence (FCS). The data in the MPDU follows the same order as the PPDU: the least significant bits are left in the frame and are transited first.

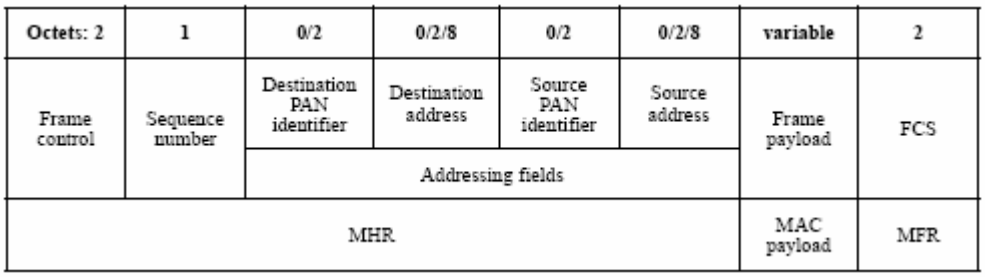



Figure 14. General MAC frame format: IEEE 802.15.4 standard, (IEEE, 2003)

The following figures present the four different MAC frame types. The beacon frame is transmitted periodically by the PAN coordinator. It provides information about the network management through the superframe and GTS fields, which are analyzed later in the section. It also synchronizes the network devices and indicates the proper communication period for them. The data frame payload encapsulates data from the higher layers. When a device receives a packet, it is not obliged to response with an

acknowledgement packet. Finally, the command frame identifier and command payload fields of the command frame are used for communication between the network devices. The command identifier specifies actions like association, disassociation, and data, GTS or beacon request.

| Octets: 2 | 1 | 4/10 | 2 | variable | variable | variable | 2 |
|---|---|---|---|---|---|---|---|
| Frame control | Sequence number | Addressing fields | Superframe specification | GTS fields (Figure 38) | Pending address fields (Figure 39) | Beacon payload | FCS |
| MHR | | | MAC payload | | | | MFR |

Figure 15. Beacon frame format (IEEE 802.15.4 standard, 2003)

| Octets: 2 | 1 | (see 7.2.2.2.1) | variable | 2 |
|---|---|---|---|---|
| Frame control | Sequence number | Addressing fields | Data payload | FCS |
| MHR | | | MAC payload | MFR |

Figure 16. Data frame format (IEEE 802.15.4 standard, 2003)

| Octets: 2 | 1 | 2 |
|---|---|---|
| Frame control | Sequence number | FCS |
| MHR | | MFR |

Figure 17. Acknowledgement frame format (IEEE 802.15.4 standard, 2003)

| Octets: 2 | 1 | (see 7.2.2.4.1) | 1 | variable | 2 |
|---|---|---|---|---|---|
| Frame control | Sequence number | Addressing fields | Command frame identifier | Command payload | FCS |
| MHR | | | MAC payload | | MFR |

Figure 18. Command frame format (IEEE 802.15.4 standard, 2003)

In the LR-WPAN, every PAN has its own coordinator. The PAN coordinator manages the communication in the local area; it has two options, to use or not use the superframe structure. The superframe (Figure 19) uses network beacons. If the coordinator does not want to use a superframe structure, it suspends the beacon transmission. However, the beacon is important for device association and disassociation. If the coordinator wishes to maintain close communication control in the PAN, and to support low-latency devices, it usually uses the superframe. A superframe determines a specific time period; beacons bound it. The beacon is transmitted in the first of the sixteen equal time slots that the superframe has. It is used to "synchronize the attached devices, to identify the PAN, and to describe the structure of the supeframe." The superframe can have active and inactive periods. All the communications have to be finished inside the superframe period. In the inactive periods the devices can switch to the sleep mode, but they have to be ready for the next beacon. The active period is further divided into a Contention Access Period (CAP) and a Contention Free Period (CFP). During the CAP, any device can communicate, competing with the other devices in the PAN using slotted CSMA-CA. The CFP contains GTSs and follows the CAP. A CFP may maintain up to seven GTSs and each GTS can reserve more than one time slot. However, the CAP should always be sufficient to allow new devices t join the PAN. The GTSs are allocated to specific devices. All the transactions must be completed inside the assigned time period (CAP, GTS, CFP) (IEEE 802.15.4 standard, 2003).

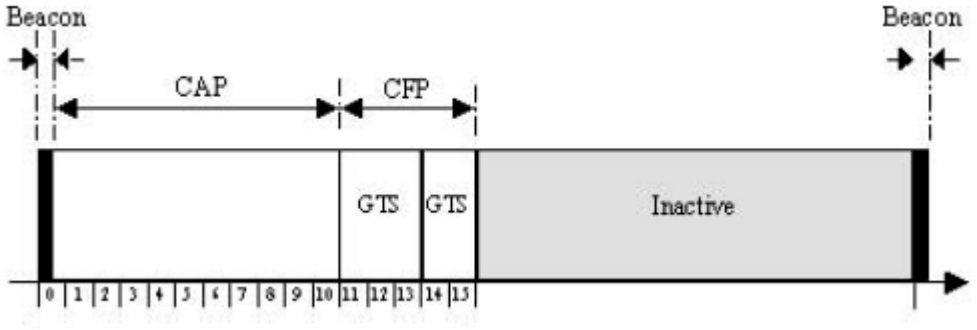
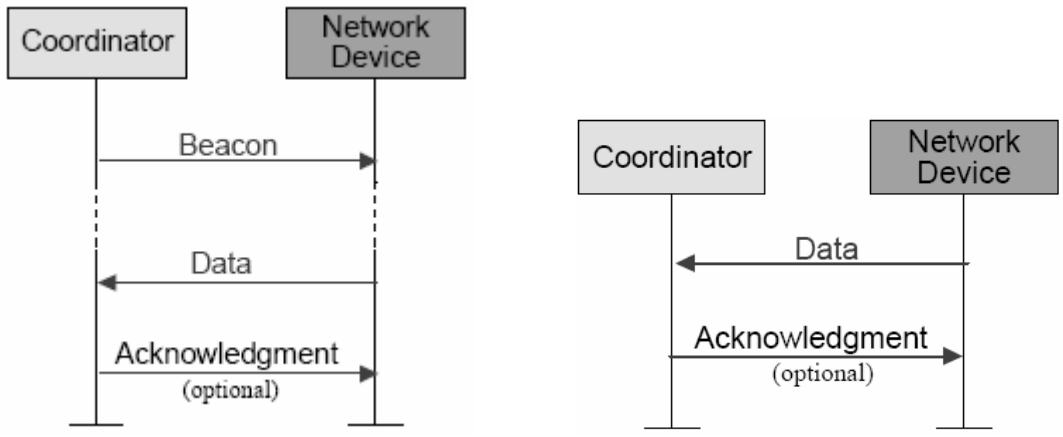



Figure 19. Example of a superframe structure (IEEE 802.15.4 standard, 2003)

In accordance with the 802.15.4 standard (2003), three different types of data-transfer exist. In addition, the types differ if the coordinator uses or does not beacons. Data transfer from a device to the PAN coordinator is the first type (Figure 20). For a "nonbeacon-enable network," it first senses the medium by using "unslotted CSMA-CA" and then a simple transmit to the data frame. In a "beacon-enabled network," the sender waits for the beacon; when it finds it, "the device synchronizes to the superfame structure." In the specified time frame, the sender again senses the medium by using "slotted CSMA-CA" and transmits the data to the coordinator. In either case, the coordinator has the option to acknowledge or not acknowledge the data reception; after that, the transaction is completed.

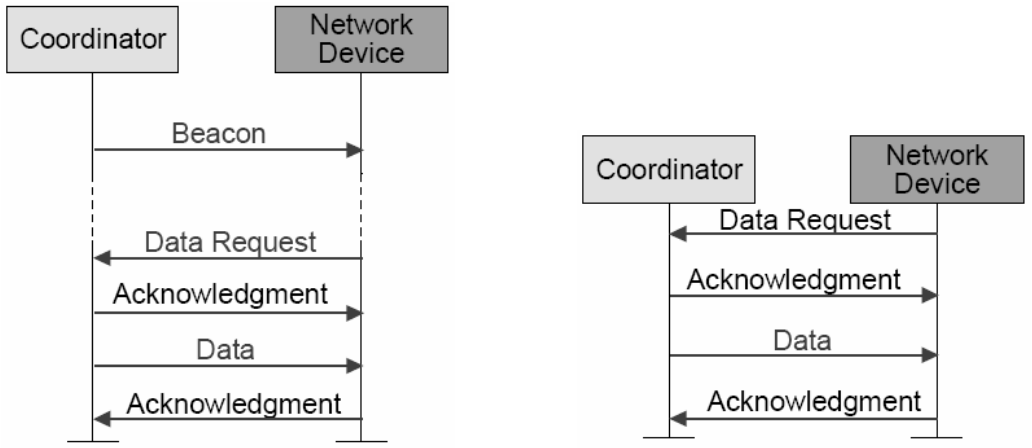



(a) Beacon-enabled network (b) Nonbeacon-enabled network

Figure 20. Communication from a device to a PAN coordinator in (a) a beacon-enabled network, and (b) a nonbeacon-enabled network (IEEE 802.15.4 standard, [IEEE, 2003]).

Data transfer from the coordinator is the next type described in the standard (2003). In a beacon-enabled network the coordinator indicates a pending message through the beacon. The message's target device receives the beacon and if the message is pending, it responses with a MAC command request message, using slotted CSMA-CA. the coordinator may or may not acknowledge the command message and

through slotted CSMA-CA sends the pending message. The device acknowledges the received message. After that, the coordinator removes the message from the beacon's pending list and completes the transaction. In a nonbeacon-enable network the device periodically sends a MAC command-request frame to the coordinator. The coordinator acknowledges the data request. Then, by using unslotted CSMA-CA, if it has a pending message it transmits it. If it has not it respond with a data frame with a "zero-length payload." To complete the transaction the device acknowledges the data reception.




(a) Beacon-enabled network (b) Nonbeacon-enabled network

Figure 21. Communication from a PAN coordinator to a device in (a) a beacon-enabled network, and (b) a nonbeacon-enabled network (IEEE 802.15.4 standard [IEEE, 2003])

Peer-to-peer is the last type of data transfer. In this situation the devices are free to communicate with any other device within their communication range. In a peer-to-peer PAN the devices can "either receive constantly or synchronize with each other." If they are receiving constantly, to transmit data they use unslotted CSAM-CA. In the second case, synchronization must be achieved first (IEEE 802.15.4 standard [IEEE, 2003]).

The IEEE 802.15.4 (2003) standard establishes MAC and PHY standards for low-cost, low-power, and high-density node deployments. In addition to the above

PHY and MAC characteristics, IEEE 802.15.4 provides a security baseline, including "the ability to maintain an Access Control List (ACL) and use symmetric cryptography" for data encryption. The algorithm that is used for encryption is the Advance Encryption Standard (AES). However, the higher level layers decide when security is need. The upper layers are in general responsible for device authentication and key management. The next section introduces the ZigBee standard, which encapsulates the IEEE 802.15.4 and provides additional standardization for the higher levels.

## 2. ZigBee

Heily (2004) defines ZigBee as "a rapidly growing, worldwide, non-profit industry consortium" whose mission is "to define a reliable, cost-effective, low-power, wirelessly networked, monitoring and control product based on an open global standard." The following figure illustrates the areas of interest for different wireless communication standards.

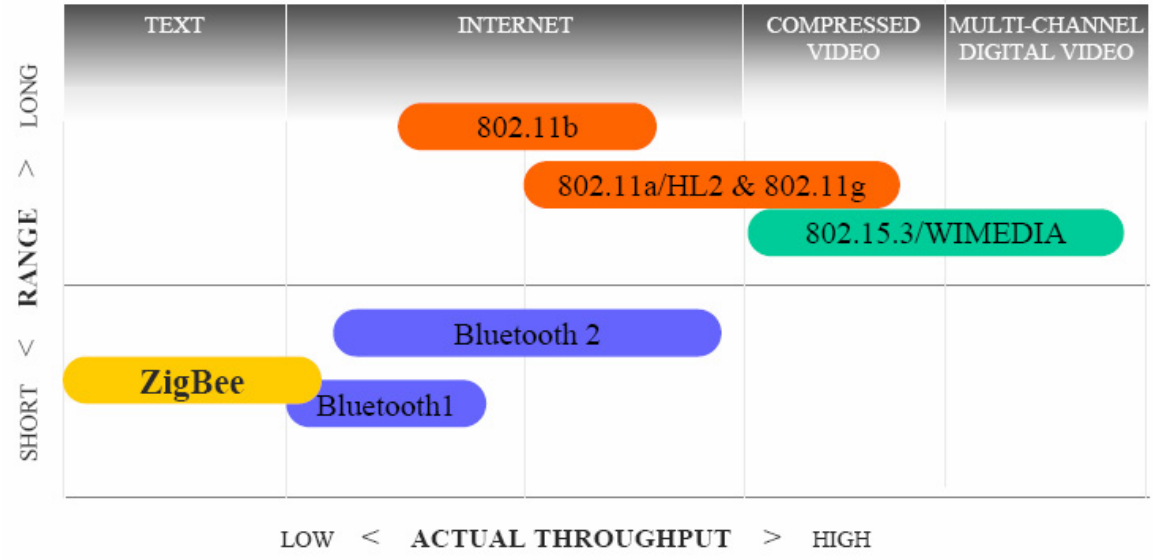



Figure 22. Overview of the coverage for different wireless communication standards (Heily, 2004)

ZigBee, a new standard which became publicly available in June 2005, is based on the IEEE 802.15.4 standard. It expands the IEEE 802.15.4 by adding the framework for "the network, security and application" (Craig, 2005). The following figure presents the IEEE 802.15.4/ZigBee stack and the areas of responsibilities.

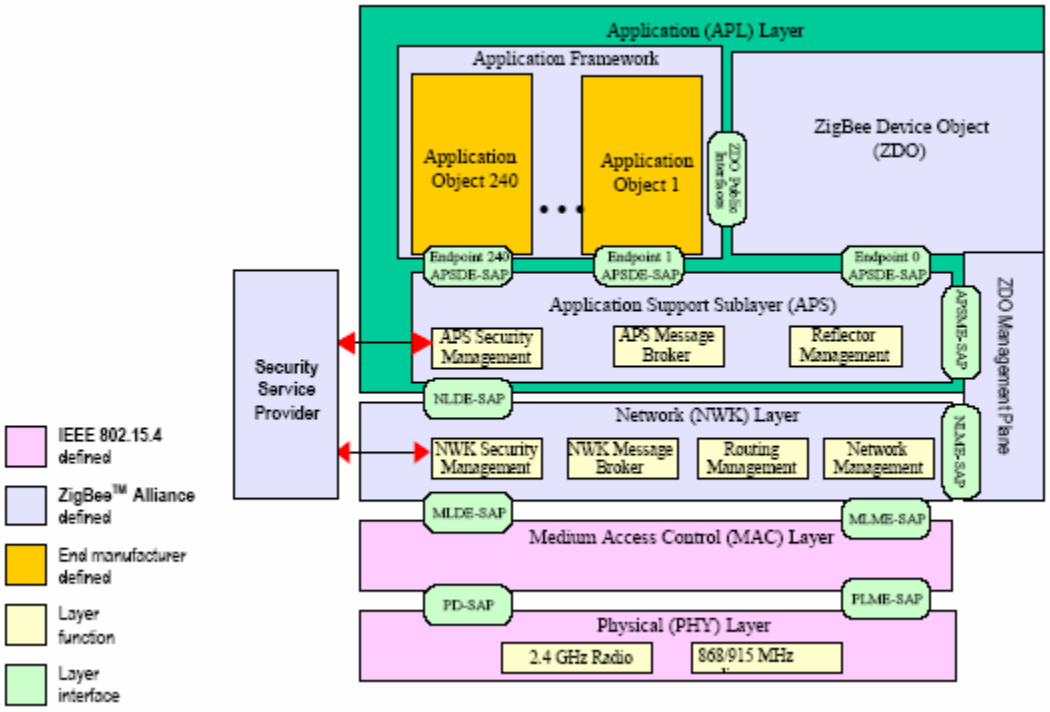



Figure 23. IEEE 802.15.4/ZigBee stack (ZigBee, 2005)

Craig (2005) mentions three networking topologies that the standard covers: the star, mesh, and cluster tree (figure 24). The ZigBee standard works on top of the IEEE 802.15.4 addressing schema by using the standard 64-bit and the short 16-bit addressing. Kinney (2005) summarizes the ZigBee network layer responsibilities: the successful establishment of a new the network, and successful new device configuration, addressing assignment, network synchronization, frames security, and message routing.

ZigBee further distinguishes the concept of the physical devices (RFD, FFD) by using the notion of "logical devices." "ZigBee Coordinator" is the first type of logical devices. It is responsible for initializing, maintaining, and managing the network. Under the coordinator in the network hierarchy is the "ZigBee router," which is responsible for controlling the message routing between the nodes. Finally, the "ZigBee End Device" acts as the end point of the network structure. The tasks that an end device can perform are specified in the "Application Profiles" (Craig, 2005).

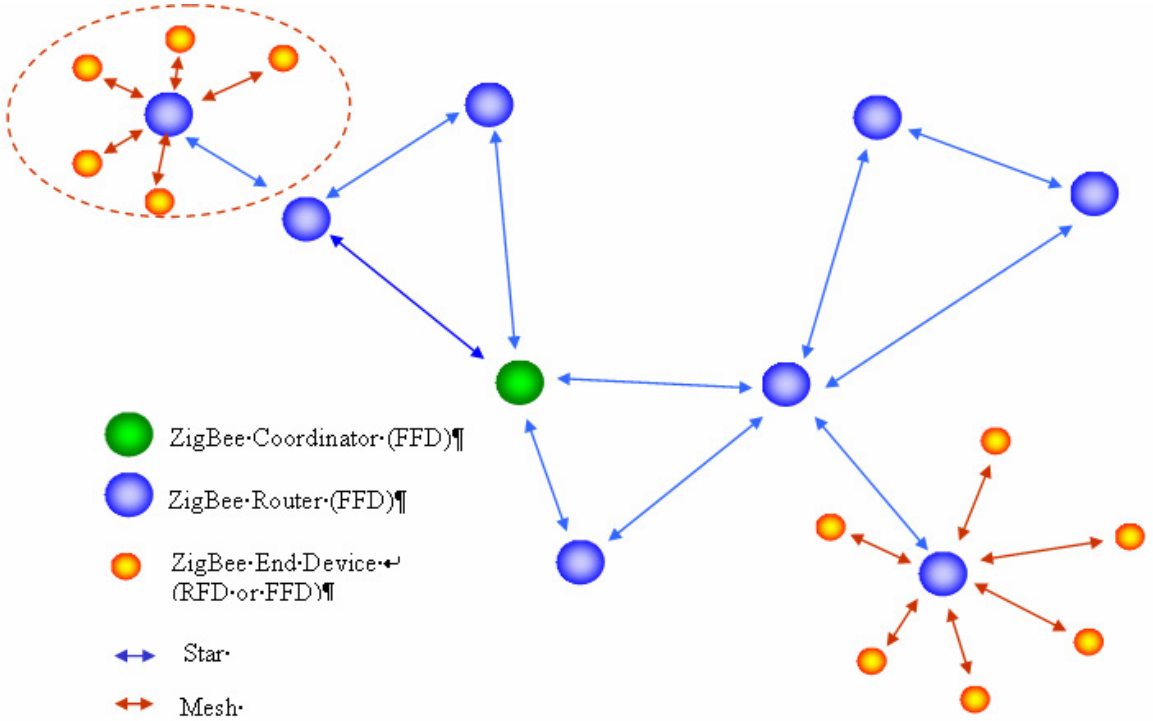



Figure 24. ZigBee network topologies (Kinney, 2005)

The ZigBee specifications (2005) summarize the security services provided by ZigBee: "key establishment, key transport, frame protection, and device management." ZigBee builds its security mechanism using symmetric key cryptography. The security services also depend on the associated layer, thus as the Figure 23 shows, the security mechanism covers the network and the application layer. In addition, if a MAC frame needs security protection, the MAC layer is able to secure it. Moreover, the notion of end-to-end security is supported; the source and destination devices have access and use the same share key.

In the MAC layer the 802.15.4 AES mechanism provides the proper security. The mechanism protects "the confidentiality, integrity, and authenticity of the MAC frames" (Kinney, 2005). An auxiliary header field in front of the MAC payload indicates if the frame is encrypted or not. The MAC frames' integrity is supported by calculating and using a Message Integrity Code (MIC) at the end of the MAC payload. In addition to the

AES, nonce is used to provide MAC confidentiality and authenticity. The following figure illustrates a MAC frame with security. For different security aspects the MAC layer uses different mode of the AES: for the encryption it uses the AES in Counter (CTR) mode, and for the integrity, the Cipher Block Chaining (CBC-MAC). Finally, the combination (CCM) of the above two modes is available, providing both encryption and integrity (Kinney, 2005).

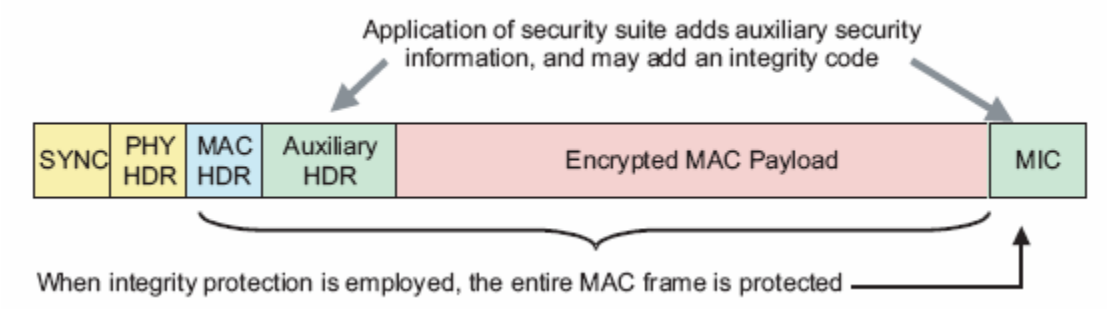



Figure 25. ZigBee secure frame in MAC layer (ZigBee specifications, 2005)

In the network layer the CCM* (a modified MAC layer CCM mode) is used for encryption. Because the network layer uses only the CCM* mode, a single key is used for all different security options. The network layer security message format is similar to the MAC frame; it is presented in the following figure. Finally, although the network layer is responsible for securing its layer messages, the above layers specify the keys and the CCM* option for each frame (Kinney, 2005).

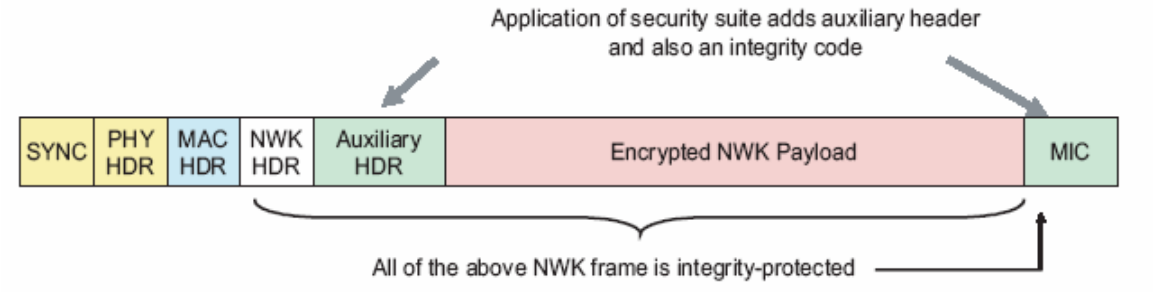



Figure 26. ZigBee secure frame in network layer(ZigBee specifications, 2005)

Security in the application layer works similar to the network and MAC layers. It uses the “link key” or the “network key” to secure the message and then encapsulate it inside a set of fields similar to the network format (figure 27). Other security responsibilities that the application layer has are to provide the ZigBee Device Objects (ZDO) and the applications with device management services, key establishment, and key transport (ZigBee specifications, 2005).

The ZigBee application layer contains the manufacturer-defined application objects, the ZDO and the application sub-layer. In addition to the security responsibilities, the application sub-layer binds devices based on their duties and needs, maintains the binding tables, and forwards messages between them. The application sub-layer also discovers the neighbor devices for a given device. The ZDO is responsible for determining the device’s duty in the network, for communicating using binding requests, and for supporting security, as was mentioned above. The sub-layer that implements the actual application is the manufacturer-defined application object (Kinney, 2005).

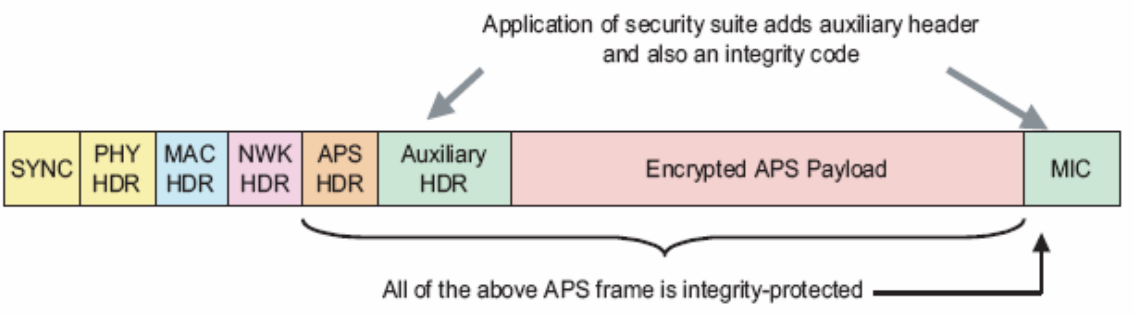



Figure 27. ZigBee secure frame in application layer (ZigBee specifications, 2005)

The ZigBee specifications are a great step toward the wireless sensors networks standardization. They cover all the OSI layers, from the physical to the application providing guidance to the developers. The specifications are the product of the ZigBee alliances, a consortium of companies and academic institutes. Thus they not only produce

the specifications, but also are the first use them. The next section goes beyond the layer architecture and standardization and introduces the operating systems (OS) field, another crucial element of WSNs.

## G. TINYOS

The above sections provided a brief description of the representative wireless sensor networks industry standards related to specific architecture layers. The purpose of this section is to introduce operating systems suitable for a WSN implementation and its nodes. There exist a number of real-time operating systems, some of which are VxWorks, WinCE, PalmOS, and QNX, but it seems that they do not meet the needs of the wireless sensor networks. A prominent solution specifically designed to satisfy WSNs' requirements is TinyOS (Hill, Szewczyk, Woo, Hollar, et al., 2000).

Similar to the traditional conventional operating systems, TinyOS provides abstractions of the physical devices. The approach is different because of the resource constraints, the application-specific implementations, the necessary modularity, and, in general, the WSNs requirements. TinyOs expresses the above abstractions using a simple component model. A number of components are used to support a particular application. The component model, in addition, uses an event-driven concurrency to satisfy properly limited resource devices, which have to process a great amount of information on the fly (Culler, Jason, Buonadonna, Szewczyk, & Woo, 2001), (Levis Madden, Gay, Polastre, et al., 2004).

A TinyOS application consists of a set of components and a scheduler. Components can be hardware abstractions, synthetic hardware, and high-level software. Each component is described by four elements: a set of commands, a set of events, a frame, and a set of tasks. All the commands, events, and tasks are executed in the context of the frame. The sets of commands and events can also be described as the component's interface to the rest of the system. Commands can be defined as non blocking requests to lower-level components to initiate an action; and they normally post a task for later execution. They can also initiate lower component commands, but those have to be completed in a short period of time. They cannot initiate events. Events represent hardware events or the completion of commands. An event can store information to each

frame, fire higher level events, call lower level commands, or post tasks. Finally, tasks are the component element that performs the main job. They are able to call lower level commands, fire higher level events and post other tasks. (Hill, Szewczyk, Woo, Hollar, et al., 2000), (Culler, Jason, Buonadonna, Szewczyk & Woo, 2001), (Levis Madden, Gay, Polastre, et al., 2004).

The two-level scheduler's hierarchy provides the TinyOS concurrency. The scheduler allows the events to preempt tasks, but tasks have to run to completion related to other tasks. Thus the desired concurrency inside a component is accomplished by the asynchronous execution of the events and tasks. The following figure illustrates a typical configuration for a networking sensor. It presents different types of components, the information flow, the commands, the events, and the event handlers. (Hill, Szewczyk, Woo, Hollar, et al., 2000), (Culler, Jason, Buonadonna, Szewczyk & Woo, 2001), (Levis Madden, Gay, Polastre, et al., 2004).

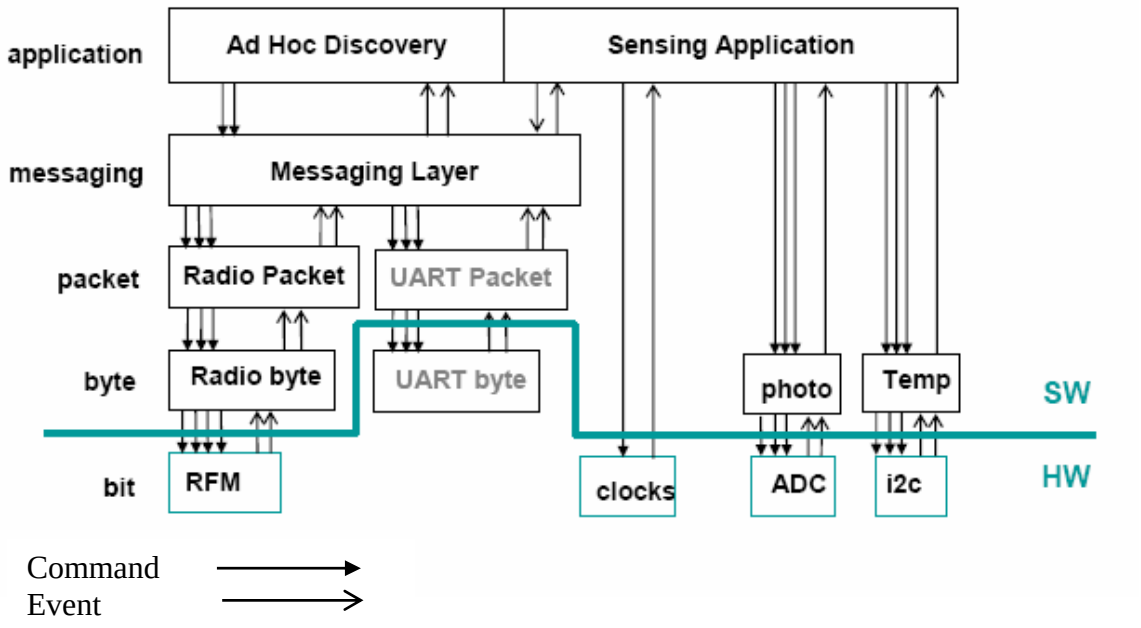



Figure 28. Typical networking application component graph (Culler, Jason, Buonadonna, Szewczyk & Woo, 2001)

The TinyOS library contains a great variety of networking applications capable of supporting a variety of wireless ad-hoc mesh architectures supporting not only single-hop routing but also multi-hop. The above application can be placed in three general categories: “tree-based collection,” in which data are routed toward an end point; “intra-network routing,” which describes data exchange between nodes inside the local network; and “dissemination where data is propagated to entire regions” (Levis Madden, Gay, Polastre, et.al., 2004). TinyOS uses the Active Messages (AM) concept to support the networking and routing functions. AM supports message-based communication and is also used in parallel and distributed computing systems. “The lightweight architecture of Active Messages can be leveraged to balance the need for an extensible communication framework while maintaining efficiency and agility.” The “event centric nature” of the AM makes them suitable for wireless sensor network applications. AM implementation “avoids busy-waiting for data to arrive and allows the system to overlap communication with other activities” (Buonadonna, Hill, & Culler, 2005).

To summarize, this chapter has provided an overview and described the evolution of wireless technology. It began with the concept of the wireless ad-hoc mesh networks and continued by introducing the new area of the wireless sensor networks, their applications, related concerns and issues, and, finally, the current WSN standards. Its intention is to provide the reader with the theoretical background knowledge that is necessary to understand the hardware and software products mentioned in the following chapters and the algorithmic applications, such as the tracking object application described in chapter IV.

THIS PAGE INTENTIONALLY LEFT BLANK

# III. OBJECT TRACKING

## A. INTRODUCTION

This chapter presents the components of a complete demonstration system, part of which is Object-Tracking. The system receives inputs from the environment, manipulates them, makes decisions, and proceeds with specific actions related to the tracking object. The purpose of this work is to demonstrate a system that is capable of being a useful plug-and-play implementation, is rapidly deployable, and is able to provide critical data to a control station far from the system's position. The system can be divided into three parts: the sensor network, the tracking object application, and the action-perform TSSR part.

The first components to be described are Crossbow's hardware and software products. They use many of the sensor network principles and ideas, already mentioned in chapter 2. They comprise the basic function of the system and provide the important initial raw data for further evaluation and manipulation. A subsequent chapter will present preliminary testing and evaluation results of the Crossbow material.

The chapter also presents the TSSR subsystem, which consists of both hardware and software parts. The subsystem detects objects by performing picture comparisons. It then transmits those pictures to the control station by using satellite or cellular communications. In the overall system, the subsystem is responsible for taking photos, acting, at the proper time after the object's detection, and tracking. In addition, it is responsible for transmitting the photos and any additional information that characterizes the object. The following figure presents a high-level view of the object-tracking application and the TSSR system.

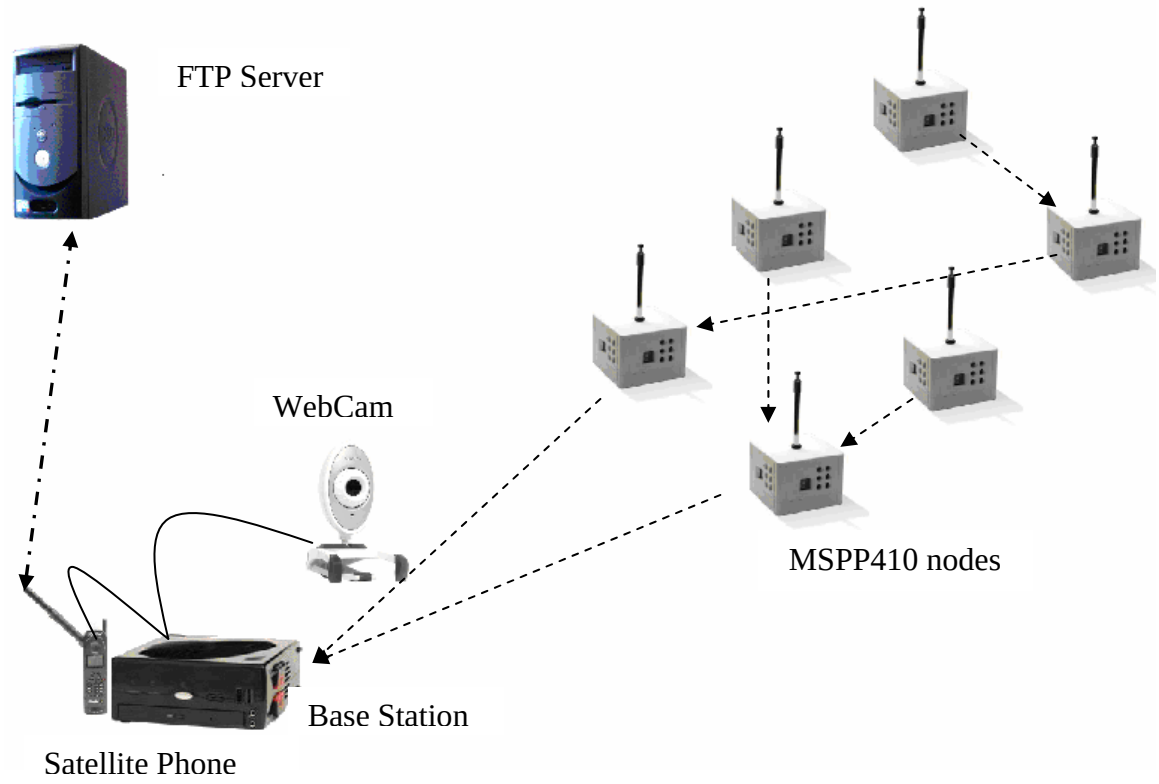



Figure 29. Overall System High-Level View

## B. OVERVIEW OF THE HARDWARE AND SOFTWARE PRODUCTS

### 1. Crossbow Overview

Before describing of the wireless sensor network system that is used by the tracking object application, it is important to provide its manufacturing profile. According to the company's web site (http://www.xbow.com), Crossbow Technology is "a leading supplier of inertial sensor systems for aviation, land, and marine applications and other instrumentation sensors as well as the leading full-solutions supplier in the wireless sensor networking arena and the only manufacturer of smart dust wireless sensors." In the last few years the company is constantly working in the wireless field providing "sensing devices and mesh networking platforms" to a great variety of

applications. By using UC Berkeley's TinyOS operating system in the produced platforms, its architecture is considered open source.

In the context of this thesis, two Crossbow sensor network solutions are used. One is the Mote-KIT2400 – MICAz developer's kit is a "Commercial Development Platform for MICAz Motes." This study uses this kit to explore the sensor network's field and to become familiar with WSN's hardware implementations. The kit provided the ability to program the nodes, by using netC, and to test different Crossbow software applications. The following paragraphs briefly present the capabilities of that system.

The description of the Crossbow material is followed by an overview of the software products that this project used. Finally, the subsequent sections present the MSP410 Mote Security System, a Crossbow commercial product and the second and main system used for this thesis.

### 2. Mote-KIT2400 – MICAz

This subsection provides a brief overview of the Crossbow development kit, Mote-KIT2400–MICAz. Understanding the available hardware is an important step for any kind of implementation. The company's webpage (http://www.xbow.com) provides an overview of the eight node kit. Mote-KIT2400–MICAz, which uses Crossbow's new processor/radio board, MICAz (MPR2400CA). The scope of the kit is primarily for demonstration and testing; but it can also be used in real-world applications like "residential and industrial building monitoring and security or in automotive networks." For that kind of use it probably needs changes, for example, the use of an outside cover for the motes and the interface board. The parts that are contained in the kit are illustrated in Figure 30 and discussed below.

Figure 30. Photo of the entire Mote-KIT2400 – MICAz (http://www.xbow.com)

***a. MICAz Processor/Radio Boards - MPR2400 (MICAz)***

Figure 31 presents one of the eight MICAz Processor/Radio Boards contained in the kit. The MPR/MIB User's manual (Crossbow, 2005) provides information about the latest generation mote, MICAz, which is compliant with the IEEE 802.15.4 and ZigBee standards. The radio frequency transceiver is the Chipcon CC2420, integrated with an Atmegal 128L micro-controller supporting wireless low-power sensor networks. It works in the 2400MHz to 2483.5MHz band (ISM band) and uses direct sequence spread spectrum techniques, which avoid RF interference and provide basic data security. By using the TinyOS, the battery-powered MICAz is compatible with other Crossbow software implementations.

Figure 31. MPR2400-MICAz with standard antenna (Crossbow, 2005)

***b. MTS300CA / MTS310CA***

The Mote-KIT2400 contains both MTS300CA and MTS310CA sensor boards (Figure 32). The details below are derived from the MTS/MDA Sensor Board User's Manual (Crossbow, 2005).

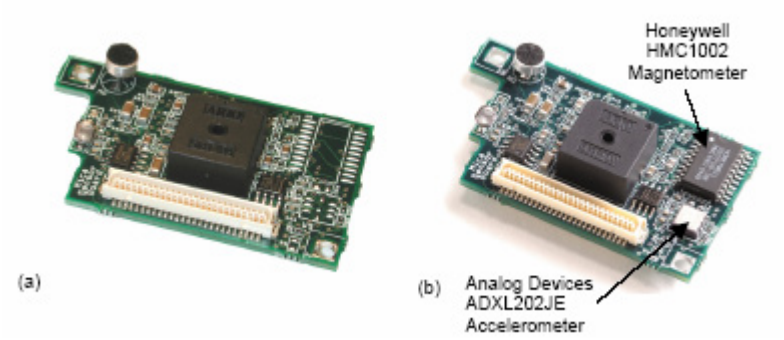



Figure 32. (a) MTS300CA and (b) MTS310CA (Crossbow, 2005)

To support a range of applications, including vehicle detection, movement, and others, these two sensor boards provide a variety of sensing capabilities. The most basic sensors are the microphone and the sounder. "Acoustic ranging and general acoustic recording and measurement" are the two main uses of the microphone. In addition, audio files can be first recorded into the MICAz flash memory and then be downloaded and analyzed. The sounder, or "buzzer," is a "piezoelectric resonator" producing a 4 KHz fixed frequency. Acoustic ranging is an application that uses the sounder and the node acoustic detector (microphone).

Light and temperature comprise another set of sensors of both MTS300CA the and MTS310CA. The maximum sensitivity of the photocell, CdSe, is at the 690 nm wavelength. The thermistor (Panasonic ERT-J1VR103J), on the other hand, provides output for temperatures from -40 to 70 Celsius degrees.

The MTS/MDA Sensor Board User's Manual (Crossbow, 2005) specifies additional sensing capabilities only for the MTS310CA. Those capabilities are the two-axis Accelerometer and the two-axis Magnetometer. The accelerometer has 10-bit resolution and is suitable for applications like "tilt detection, movement, vibration, and /or seismic measurements." The magnetometer is a Honeywell product (HMC1002) very sensitive to small magnetic fields. It can be used to detect vehicles at a radius of 15 feet.

### *c. MIB510 Serial Interface Board*

To support communication and programming with other systems the kit includes the MIB510. The MPR/MIB user's manual (Crossbow, 2005) provides the following information for this product. The interface board serves not only the MICAz

but also the MICA2, MICA, and MICA2DOT family products. In contrast to the MTS300 and MTS310, the serial interface board in not battery powered. Figure 33 is a top-view photo of a MIB150CA. The Atmega16L is the in-system processor (ISP); it runs at a fixed rate of 115.2 kbaud to support motes programming. When the program is downloaded from a PC to the ISP through a serial port RS-232, the ISP programs the mote that is connected on top of the MIB510. Both the mote and the ISP share the same RS-232. Motes programming means that, in addition to the kit, the user has to install TinyOS in the PC as an important development platform.

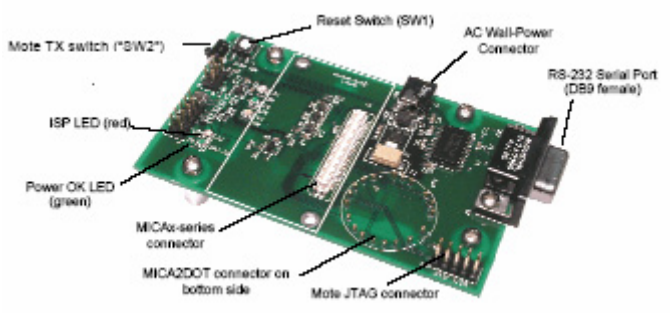



Figure 33. MIB510CA (Crossbow 2005)

Although the Mote-KIT2400–MICAz is a development kit, its study was very important. It gave an overview of a sensor network implementation. Moreover, it provided an opportunity to test and improve the developer's programming skills in TinyOS and to examine the Crossbow implementations. In addition to the above hardware, the tracking object application development used Crossbow software products like MOTE-VIEW, not only with the Mote-KIT2400–MICAz but also with the MSP410 Mote Security System. The following is an overview of those Crossbow software solutions.

## 3 Crossbow Software Solutions

### *a. XMesh Network Stack*

The XMesh Network stack is Crossbow's implementation for the wireless ad-hoc mesh networks that its system uses. It was developed based on the IEEE 802.15.4 and ZigBee standards mentioned in previous chapters. Unfortunately, as a proprietary protocol, the exact specifications were not available at the time this project took place.

### *b. MOTE-VIEW Client Software*

In the MOTE-VIEW 1.0 User's Manual (Crossbow, 2005) Crossbow says that MOTE-VIEW is an important element of its software solutions. The importance of MOTE-VIEW is precisely described in the definition as Crossbow's primary user-interface, sited between the user and an already deployed wireless sensor network. Its purpose is to make the deployment and monitoring of the system easier. Additionally, it supports wireless sensor data logging to a database, analysis, and presentation of those data.

The MOTE-VIEW user's manual image presented in Figure 34 shows a complete three-layer wireless sensor network implementation architecture that part of it is MOTE-VIEW. The client layer is where MOTE-VIEW is located, providing monitoring interpretation and analysis of the raw data returned by the sensors. Those data arrive at MOTE-VIEW through the second-server layer where they are first stored in a database for logging purposes. Finally, in the mote layer, motes use the onboard sensor, and through their program written in TinyOS, perform a specific task, gathering the proper data for the application.

Crossbow claims that MOTE-VIEW supports all the company's wireless sensor network hardware. Initially, in the research part of this study, MOTE-VIEW was used with the MICAz-based Mote-KIT2400. Later, throughout the first steps of the tracking object application construction, MOTE-VIEW was used heavily, first for understanding the MSP410 system's topological and networking functions, and then for investigation of the returning data. Additionally, MOTE-VIEW was used constantly during the experiments for wireless sensor network system setup, monitoring, and evaluation. When it was important, the database storage ability that it provides was used. Finally, instead of the RS-232 an optional-part of the tracking object implementation reads the sensors data from the MOTE-VIEW postgress database. This option requires MOTE-VIEW installation in the system's control station.

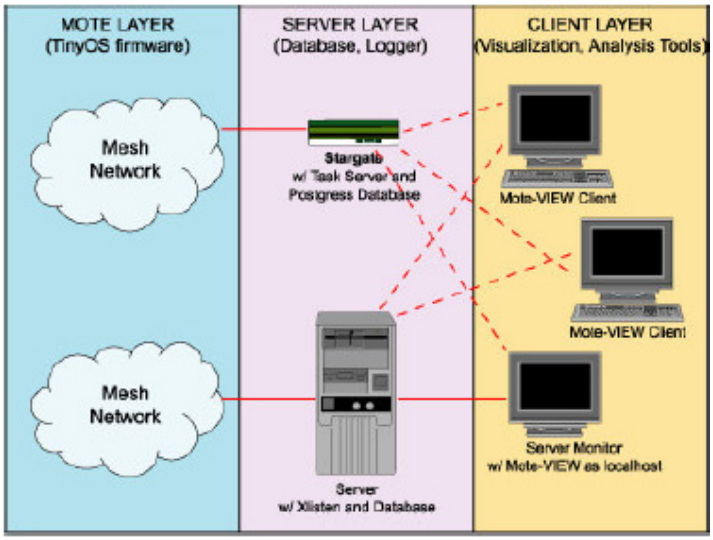



Figure 34. Three-layer software framework for a wireless sensor network: MOTE-VIEW 1.0 User's Manual (Crossbow 2005)

As mentioned above the MOTE-VIEW's primary objective is to provide an easy, and functional graphical user interface. The following sections provide a brief description of the interface's functionality, as explained in the MOTE-VIEW user's manual (Crossbow, 2005).

Once a sensor network is setup and connected to the user's computer, which runs MOTE-VIEW, the user has the ability, after the proper database and firmware application configuration, or after starting a "data log" menu option, to observe data from the database. Figure 35 presents a screenshot of data returning from the MSP410 system. Through the window, "data view" tab selection, the user is able to watch the sensor's data, the nodes status, and server's possible messages.

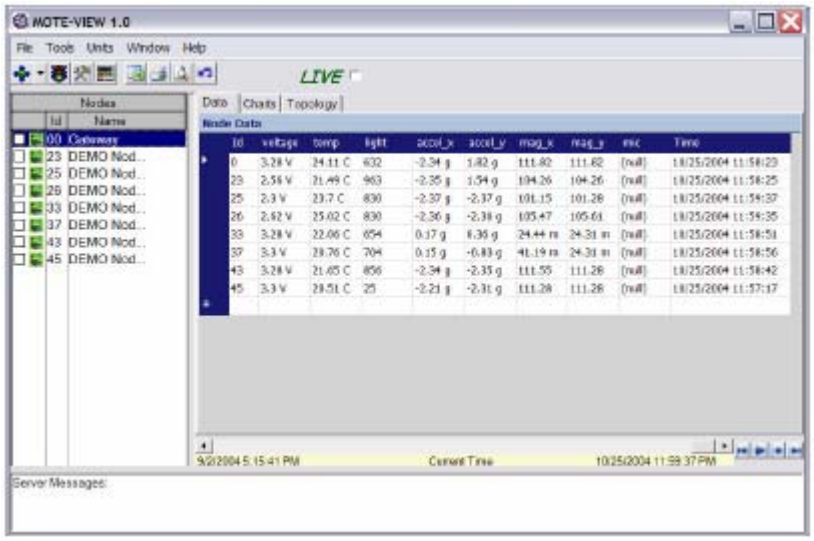



Figure 35. Screenshot presents MOTE-VIEW "Data" view received from MSP410 system: MOTE-VIEW 1.0 User's Manual (Crossbow 2005)

Additionally, the user can examine different aspects of the sensor network system by selecting different menu options. The "Chart" tab provides the ability to produce sensor's historical data graphs: Figure 36 applies to the graphs that this view provides. The "Chart" selection can present up to three different graphs from different sensors, and up to twenty-four nodes can be selecting for plotting.

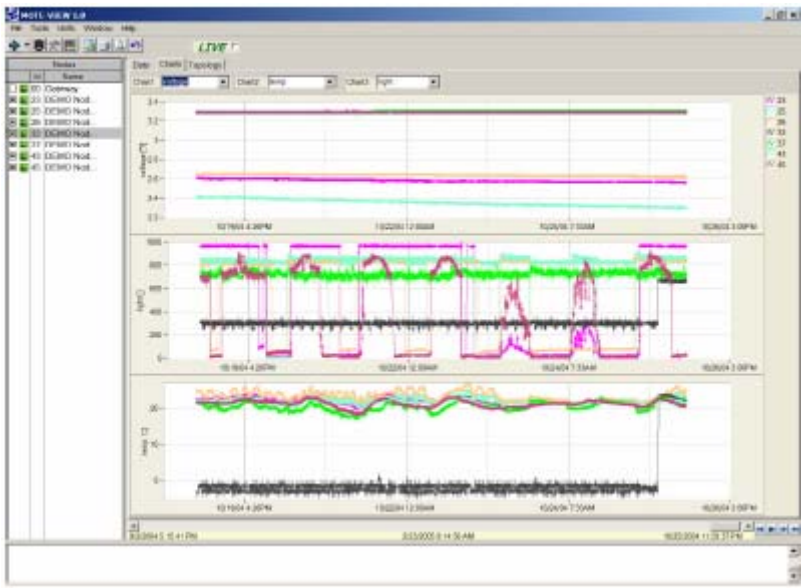


Figure 36. Screenshot presents THE MOTE-VIEW "Chart" view received from the MSP410 system: MOTE-VIEW 1.0 User's Manual (Crossbow 2005)

Figure 37 presents the final MOTE-VIEW presentation option, the "Topology," which is a drag-and-drop application that gives the user the ability to map the network's nodes, including position and parenting information. Moreover, the user has the ability to insert background images, presenting properly the system's real development environment. All of the above three charts can be printed by using the print option.

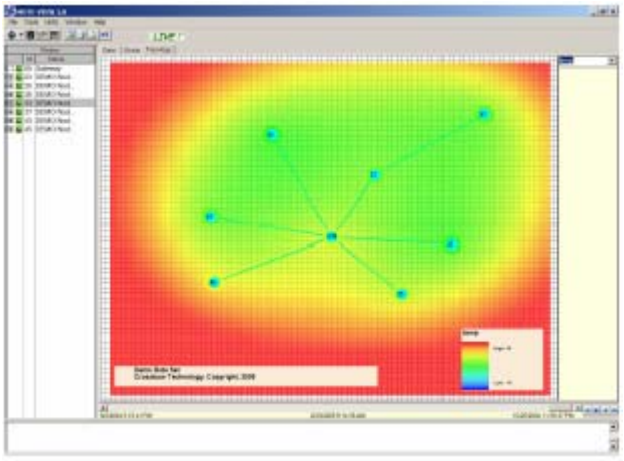


Figure 37. Screenshot presents MOTE-VIEW "Topology" view received from MSP410 system: MOTE-VIEW 1.0 User's Manual (Crossbow 2005)

In addition, MOTE-VIEW can export the received data in two different formats, XML or CSV (Comma Delimited Text). Finally, through "MoteConfig" the user can program the motes. Actually, the user does not program the motes directly; "MoteConfig" is a graphical interface that can be used to download pre-compiled TinyOS applications. Thus, in addition to the convenience of "MoteConfig" it saves the user from installing the TinyOS programming environment. However, full control and downloading capabilities are provided only by using TinyOS.

***c. XServe***

XServe, as described in the MOTE-VIEW 1.0 User's Manual (Crossbow, 2005) is a command-line tool that facilitates the sensor's data readings. The user can use XServe from a *Cygwin* command line, or as a data-logging server for MOTE-VIEW, using the *LogData* menu.

### *d. Surge Network Viewer (Surge-View)*

Surge-View is another set of software tools provided by Crossbow. It contains the Surge Graphical User Interface (GUI), the Stats, and the HistoryViewer programs. Although the user is able to see the sensors' board data through Surge, this tool is mostly related to the system's networking issues. Through the GUI, the user is able to view the mote's connectivity and routing statistics. Additionally, the network performance can be stored in the control station (PC) for later usage. Stats provides data about the network's condition. Finally, HistoryViewer enables the network's topology and statistics playback. The following figures illustrate different outputs of the Surge-View software product. (Getting started Guide, Crossbow, 2005).

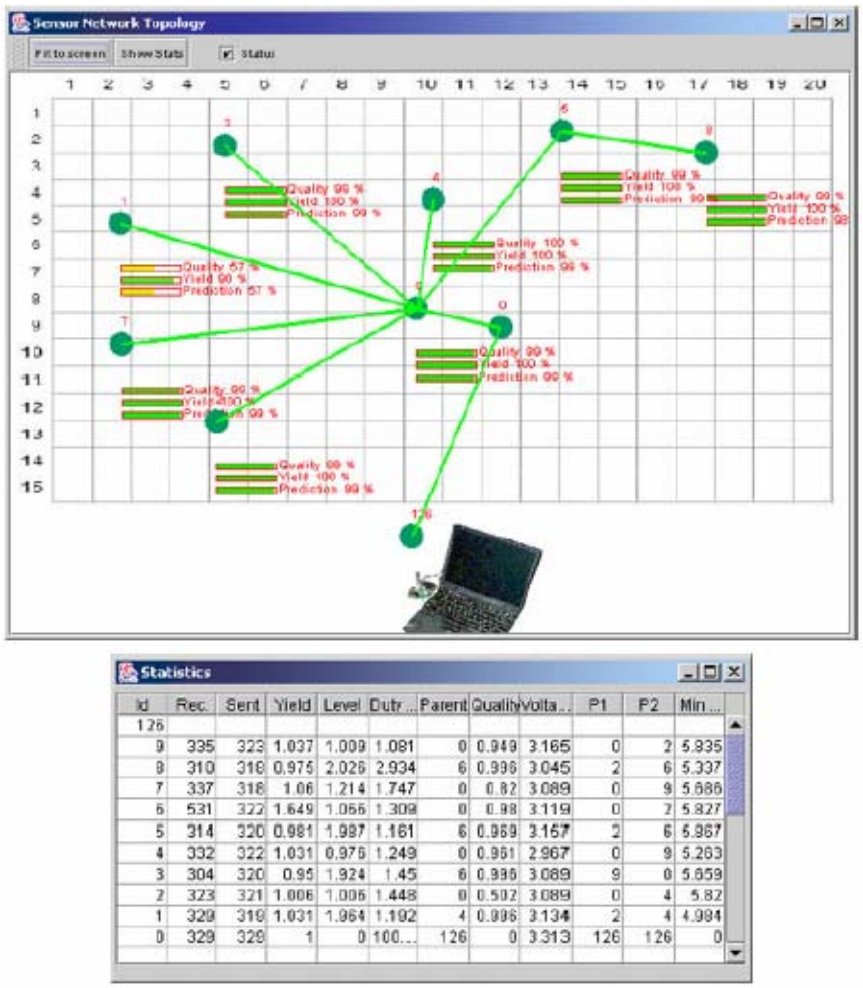



Figure 38. Surge's output for a Wireless Sensor Network Topology and Statistics: Getting started Guide (Crossbow, 2005).

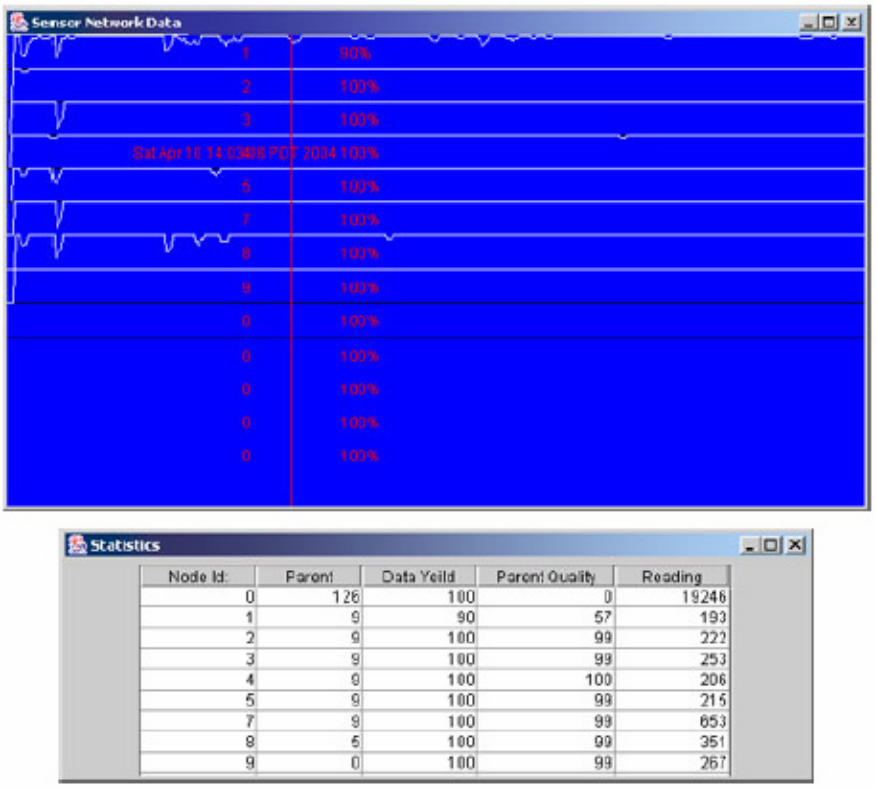



Figure 39. HistoryViewer output for a Wireless Sensor Network Data Topology and Statistics: Getting started Guide, Crossbow, 2005).

## 4. MSP410 Mote Security System

### *a. Overview*

The Mote-KIT2400–MICAz was the first wireless network system that this project worked on. Moreover, the project repeatedly used MOTE-VIEW and the rest of Crossbow's software products. However, the most important Crossbow element of the tracking object application is the MSP410 Mote Security System, a battery-powered eight-node kit targeted at serving security implementations. The MSP410 is the wireless sensor network component in the tracking object application that is responsible first for creating and maintaining the wireless ad-hoc mesh network and then for collecting and returning to the base station the sensor's values that is critical for the application. In the following sections the MSP410 system is analyzed beginning with the proposed Crossbow implementations and then providing hardware and software specifications. Figure 40 provides a high-level view of the kit.

Figure 40. High-level view of Mote Security System Deployment Overview (MSP 410)

***b. Proposed Deployments***

The MSP410 Mote Security System supports a variety of security applications. Although the object tracking is not included in the MSP410 kit (Crossbow, 2005), it is designed to support security applications using motion detection such as "remote border security, perimeter protection, intrusion detection and identification, and building occupancy monitoring."

In a typical security application, MSP410 Motes are deployed in a perimeter or grid pattern. The MSP410 mote, by combining wireless mesh networking technology and carrying a set of sensors, is able to generate detection by transmitting the proper sensor's data to the base station directly or through the network. Figure 41 presents a possible perimeter deployment around a building and exhibits the distances between the motes which recommended by Crossbow. Figure 41 also demonstrates the orientation restriction recommended by Crossbow, which the motes must have for better results. In the MSP410 Sensing Subsystem section we will further analyze this constraint.

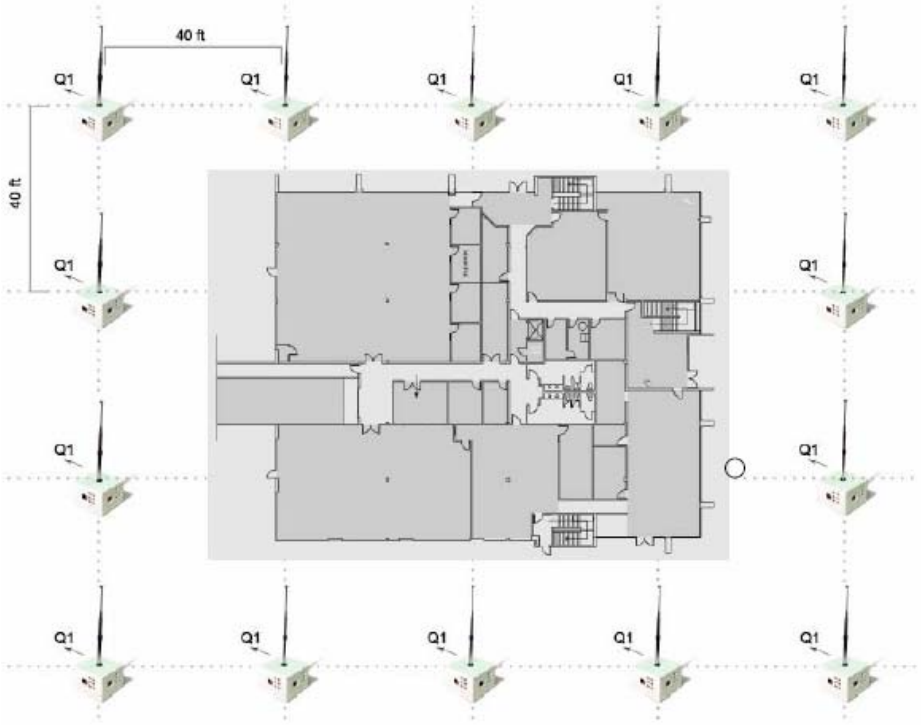



Figure 41. MSP410 deployment for perimeter monitoring: MSP410 Series User's Manual (Crossbow 2005)

The proposed dense grid deployment is presented in the user's manual and Figure 42. It includes the recommended distances and the same orientation restriction with the perimeter option. From the deployment description we assume that the purpose of the above dense grid is to provide complete coverage of the area of interest. The distances in both proposed deployments are restricted by the average sensor's effective distances and not by the communication ranges. If the application's requirements do not specify complete area coverage, then greater distances can be used between the motes. Thus a greater area will be covered by the same number of motes, but possibly, some shadow areas also will be produced.

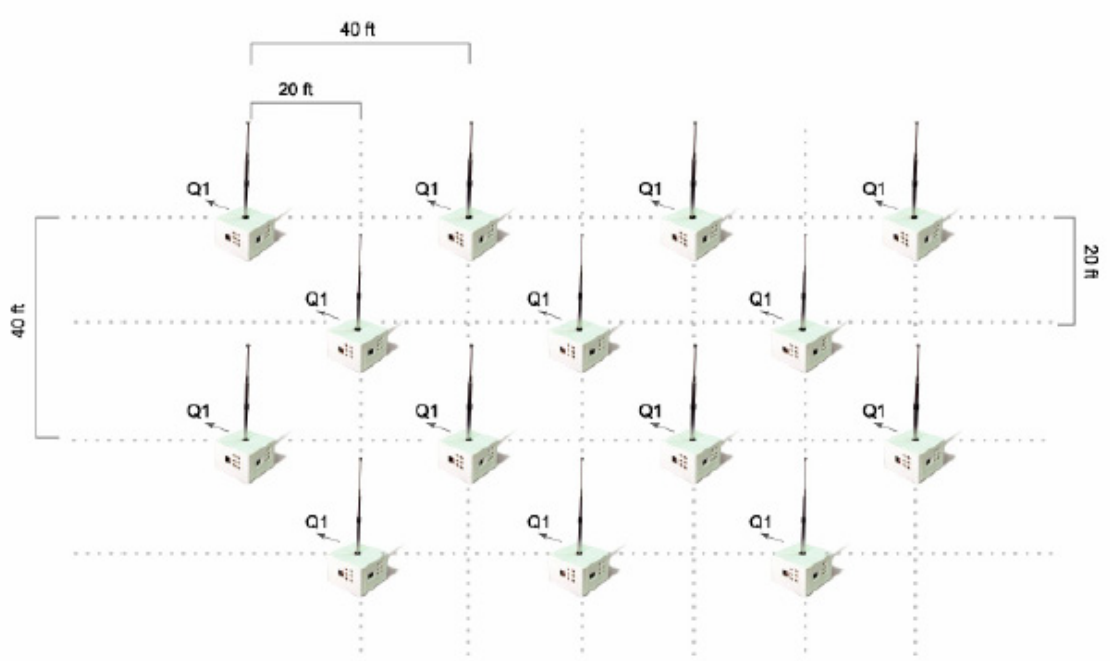



Figure 42. MSP410 deployment for a dense grid monitoring: MSP410 Series User's Manual (Crossbow 2005)

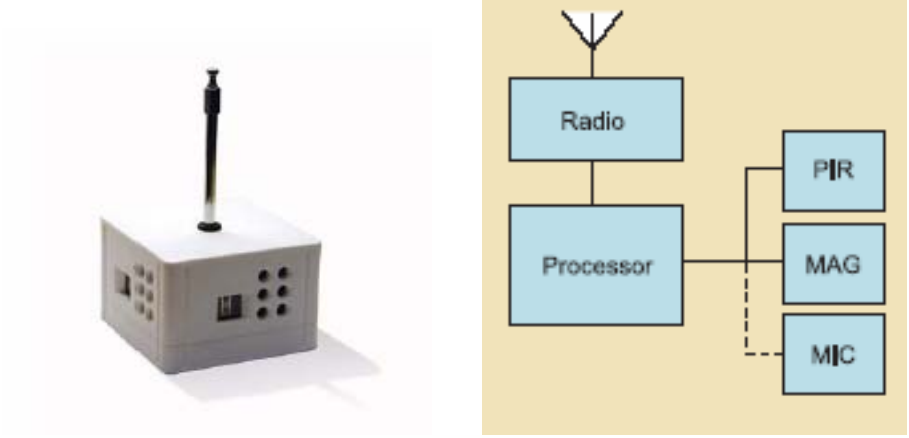
### *c. Systems Components*

The MSP410 Mote Security System can be roughly divided into two parts. One part comprises is a number of MSP410CA Motes, a two AA battery-powered "integrated processor-radio-sensing device." The mote is the system's core, responsible not only for the sensing functions but also for the deployment and maintenance of the wireless mesh ad-hoc network. Moreover, the system contains an MBR410CA, the base station, which acts as the important wireless sensor network interface with other systems. It is responsible for delivering the collected data to the connected system; it is also used to reprogram the motes, an important function for a developer or during the system's maintenance. The following sections analyze further the above system's components, based on the information included in the MSP410 Series User's Manual and the MPR/MIB User's Manual (Crossbow, 2005).

### *d. MSP410CA (mote) MICA2 Platform Core (Microcontroller, Radio)*

The core element of the MSP410CA system, the mote, is a combination of the MICA2 processor/radio board and a variety of sensors. Figure 43 exhibits the mote in

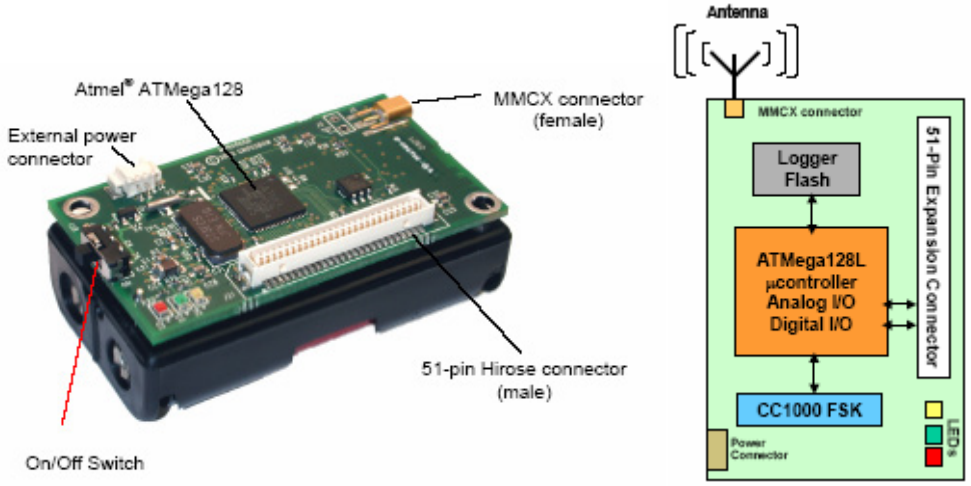
the “heat reflective plastic enclosure” and the mote’s basic block diagram. This section focuses on the platform, microcontroller, and radio.




(a) (b)

Figure 43. (a) Mote’s high level view and (b) Mote’s basic block diagram MSP410_Datasheet (http://www.xbow.com)

We begin our description at the left part of the above block diagram, the processor/radio part. The processor/radio part belongs in the MICA2 Crossbow products’ family. The MPR/MIB User’s Manual separates MICA2 into three models based on their RF frequency band: the MPR400 (915 MHz), the MPR410(433 MHz) and the MPR420 (315 MHz). The MST410 system uses the second model, the MPR410. All the MICA2 models are compatible and can communicate with each other. Figure 44 demonstrates the platform and the block diagram of the MICA2.




(a) (b)

Figure 44. (a) Photo of a MICA2 (MPR4x0) without antenna, (b) MICA2block diagram of a MPR/MIB User's Manual (Crossbow, 2005)

The basic platform element which has total control of the functions, is the Amtel Atmega128 microcontroller. Although only two peripherals are directly connected to the processor, the external flash and the 64-bit Serial ID number, all the sensors and devices are handled as peripherals. The wired communication and reprogramming functions are provided by the 51-pin Hirose interface connector. In addition the platform provides status indicators by using three different LEDs: yellow, green , and red (MSP410 Series User's Manual [Crossbow, 2005]). However, it is quite difficult to identify the mote's status using the LEDs, because of the enclosure.

The Chipcon CC1000 radio is the other vital piece of the MICA2 Platform Core. It manages transmission at an effective baud rate 19.2 kilobits per second (kbps) by using two-tone Frequency Shift Keying (FSK) modulation and Manchester encoding (MSP410 Series User's Manual [Crossbow, 2005]). Within the specified band, around 433MHz, the radio can be tuned up to four different channels. The actual number of possible channels is higher, but the recommended channel spacing in order to avoid interference is greater than 500 kHz. Furthermore, the transmission power can be adjusted by reprogramming the proper register in the radio that controls the RF power

(MPR/MIB User's Manual [Crossbow, 2005]). The default working frequency for the MSP410 system is 433MHz, and we assume that the default power level is the maximum.

The operating system that the MICA2 board runs is TinyOS 1.1.7 and higher. Additionally, the software suit includes Crossbow's XMesh networking Stack (MSP410 Datasheet [http://www.xbow.com]). The platform combines four different elements to promise reliable security application's deployment with area coverage, depending on the application, from 1,000 sq. ft. to 30,000 sq. ft. per mote. The first is platform's mesh networking capabilities. Then the effective system's deployment radio ranges, Crossbow in the MSP410 Series User's Manual claims "at least 250ft on flat concrete ground and 150ft when placed on grassy terrain with rolling hills". The last one is the sensor's capabilities, that are described in the next paragraph.

#### *e. MSP410CA (mote) Sensing Subsystem, Passive Infrared (PIR) Sensor*

Under the plastic enclosure of the MST410CA mote, except the core's board microcontroller and the radio components, is a set of sensors. These sensors are the system's important interfaces, with an environment responsible for gathering data. The collected information is passed to the board part where a basic manipulation takes place. Then the information is encapsulated in a message, which is transmitted through the network forward to the base station.

The MSP410 Series User's Manual (Crossbow, 2005) contains details related to the mote's sensing capabilities. Each MSP410 node contains a microphone that currently is not used and a set of magnetic field and passive infrared (PIR) sensors. The PIR sensor provides 360-degree coverage in a horizontal direction; to do so it uses four PIR sensing elements arranged orthogonally. Each element is considered a "dual element sensor" designed to detect the thermal that a body or object radiates. A lens enhances the sensor's capabilities; it generates a vertical field of view of ± 15° and ± 45° in the horizontal plane. The horizontal field of view is further subdivided into nine individual beans. Object detection is taking place whenever a "shadow" produced by a warm object close to a sensing element crosses sequentially at least two of the horizontal beans. The four PIR elements also provide "Quad Detect capability". This capability enhances the

system's ability to identify an object's movement and direction by including into the data message, which the node sends to the base station, not only the pir value but also the quad that had the detection.

The outputs of a PIR sensor are affected by the sensor's sensitivity, the sensor's position, the ambient thermal noise and the object's characteristics (type, size, distance, velocity, direction, aspect). To increase the sensing performance and to reduce the effect of the noise, sensors use filtering for the input output signal. Additionally, they eliminate the monitoring bandwidth by using "active filtering" in the area where they have the greatest sensitivity from 0.01 Hz to 15 Hz. Table 2 summarizes the specification and performance of the MSP410 PIR sensor based on the MSP410 Series User's Manual (Crossbow, 2005).

| Specifications - Performance | Value | Comments |
|---|---|---|
| Optical wavelength | 5 μm to 14 μm | |
| Optical bandwidth | 0.01 Hz to 15 Hz | |
| Field of view vertical | ± 15° ° | |
| Field of view horizontal | ± 45 | |
| Storage temperature | -55°C to +125°C | |
| Range for human detection | 30' to 40' | For Motes height ≈ 3' off the ground<br>Outdoor air temperature ≈ 7°C. |
| Range for cars detection | 50' to 60' | |
| Range for large tracks detection | 70' to 80' | |

Table 2. MSP410CA Mote PIR Sensor's specification and Performance based on the MSP410 Series User's Manual (Crossbow, 2005).

### *f. MSP410CA (mote) Sensing Subsystem, Magnetic Sensor*

The magnetic sensor is a very sensitive two-axis magnetic-field disorder detector. It is triggered by changes in the local magnetic field, which may be produced by a near-passing object. The use of proper noise-filtering algorithms and a two-stage amplification minimizes false detections and succeeds in maximum detection ranges. The

following table provided by Crossbow in its MSP410 Series User's Manual (Crossbow, 2005) summarizes the magnetic sensor's specifications.

| Parameter | Typical value | |
|---|---|---|
| Bridge resistance | 1100 ohms | |
| Field range | ± 6 gauss (Earth's field = 0.5 gauss) | |
| Sensitivity | 1 mV/V/gauss | |
| Linearity error (best fit straight line) | ± 1 gauss | 0.05% FS |
| | ± 3 gauss | 0.4% FS |
| | ± 6 gauss | 1.6% FS |
| Bandwidth | DC to 5 MHz | |
| Noise Density | 50 nVsqrt Hz @ 1kHz | |
| Resolution | 120 µgauss @ 50 Hz BW | |
| Storage Temperature | -55°C to 175°C | |

Table 3. MSP410CA Mote Magnetic Sensor's specification: MSP410 Series User's Manual (Crossbow, 2005)

### *g. MSP410CA (mote) Power Characteristics*

As part of the MICA2 family, the MSP410CA mote is designed to operate by using two-AA-battery power. This section focuses on some of the power characteristics of the motes. The practical operating voltage is 3.6 to 2.7 V; thus, theoretically, any battery combination that provides the above voltage can be used. Additionally, the MICA2 board can be powered through the 51-pin connector and the two-pin Molex connector. However, in the MSP410CA product, the last three abilities are not applicable because of the enclosure. The following table summarizes the power requirements for various operations. Finally, according to the system's manual, the two AA batteries last ten hour.

| Circuit | Mode | Current |
|---|---|---|
| PIR | Off | 1 µA |
| PIR | On | 300 µA |
| Magnetometer, per axis | Off | 1 µA |
| Magnetometer, per axis | On | 3 mA |
| Radio | Off | 1 µA |
| Radio | RX mode | 8 mA |
| Radio at 1 mW | TX mode | 16 mA |
| Processor | Sleep | 15 to 20 µA |
| Processor | Active | 8 mA |
| Serial flash memory | Write | 15 mA |
| Serial flash memory | Read | 4 mA |
| Serial flash memory | Off | 2 µA |

Table 4. Motes' power requirements for various operations based on the MSP410 Series and MPR/MIB User's Manual (Crossbow, 2005)

### *h. MBR410CA Mote Base Station*

The MBR410CA Base Station (Figure 45) consists of two different pieces that have already been described in the above sections. A MIB510 serial gateway and a MICA2 series MPR410 radio/processor board are connected together.

The base station primarily supports two different operations. First, by having the node ID 0 acts as a base station for the wireless sensor network, this configuration allows data aggregation from the nodes on a computer platform connected to the MBR410. In addition, the developer has the ability to reprogram the motes by using the RS-232 serial programming interface, as we have already mentioned.


Figure 45. MBR410CA, MSP410 base station

## C. TSSRV3

### 1. Overview

The main objective of this thesis develop the object detection and motion estimation application using the above described Crossbow product to gather data from the environment where it is deployed. The Object Tracking application algorithmically processes the returned data from the Crossbow wireless sensor network system and produces outputs about the object's movement. The outputs can be further used by any other system able to handle them.

The related system that is used during this project for testing and demonstration purposes is the Tactical Remote Sensor System version 3 (TSSRv3), which is briefly described in the following section. The TSSRv3 is part of the thesis research by Brian Dixon and William Felts (TNT report, 2005). It is a wireless sensor network system developed to provide data gathering, images, from the environment where it is located. Then it employs effective resource aggregation to maximize the usage of network devices and resources. By providing a capability for load sharing between the system's nodes, it further improves the resources handling. In addition, the TSSR's nodes can discover and evaluate different communication options from a predefined pool of options, trying to increase the data transmission performance. Finally, the high-quality images that the system is able to capture through satellite communication are accessible from any computer connected to the Internet. The software package is written in C Sharp (C#) and runs on top of the following hardware.

## 2. Hardware

The TRSSv3 system currently includes three sets of a 4XEM Elite miniPC, Globalstar Phone, and Creative WebCam. In addition, the system includes one File Transfer Protocol (FTP) Server. The following figure provides a high-level overview of a TRSSv3 set, including the FTP server.

(a) 4XEM Elite2 miniPC (b) Creative WebCam

(c) FTP Server (d) Globalstar Satellite Phone

Figure 46. TSSRv3 Hardware Components (a) 4XEM Elite2 miniPC, (b) Creative WebCam, (c) FTP Server, (d) Globalstar Satellite Phone (TNT report, 2005)

As a wireless sensor network system, theTRSSv3 compromises mobility, size, and power management issues. The miniPC choice, as the node main part, provides enough processing power and storage capacity for the image gathering and processing task. Additionally, the mobility requirement is supported first by making the miniPC battery powered and then through the networking flexibility and satellite communications capabilities of the system. The satellite modem and the Globalstar satellite phone serve the purpose to transmit color images from the remotely located sensor devices to the FTP server. The commercial 1.3 megapixel WebCam compromises the important image quality and keeps the image file-size small enough. Finally, to keep the system's cost

low, all the hardware components are commercial. The following section presents the system's architecture and provides an overview of the system's software part.

## 3. System Architecture

The TRSSv3 system's design as a wireless sensor network can be divided into two parts, the node-sensor and the network. The TSSR3 node is the set of a miniPC, equipped with a WebCam and a satellite phone. The hardware overview paragraph presented the node's components. The TSSR3 node software functionality is described in the following sec. tion

The network's characteristics and topology comprise the second important part of the TSSRv3 system. The system provides satellite communication with the FTP server for data transfer and uses 802.11b wireless technology for the local ad-hoc network. The purpose of the local wireless network is to implement load sharing among the system's nodes. The network topology used for the ad-hoc network is "star." The star topology decision was based on simplicity and reduced-overhead factors. The single point of control that star provides serves the simplicity factor, but it also has the drawback of the single point of failure. The system's nodes behave only as either server or client, not both; in particular, the center node is the client and the connected nodes are the servers.

Every TSSRv3 node can transmit images through the Globalstar satellite phone when the images arrive at the Globalstar ground station. Then, through standard Internet routing, they are forwarded to the FTP server. The following figures present an overview of the TSSRv3 network topology.

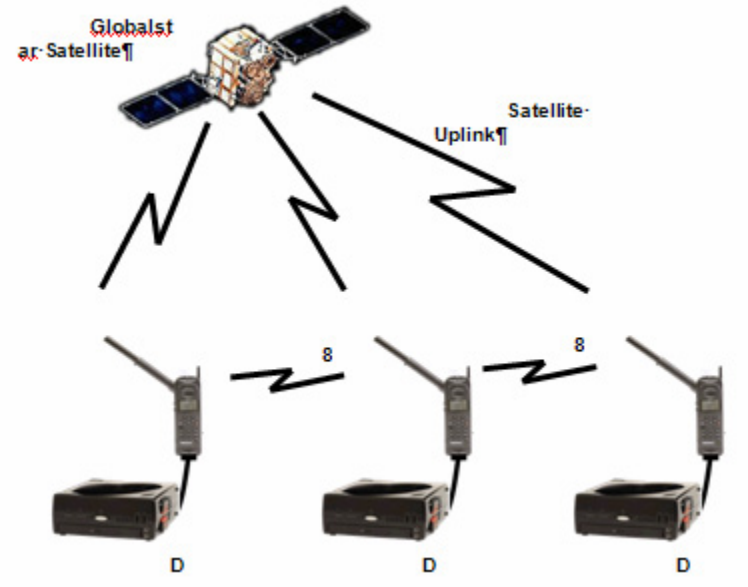



Figure 47. TSSRv3 ad-hoc network and uplink connectivity (TNT report, 2005)

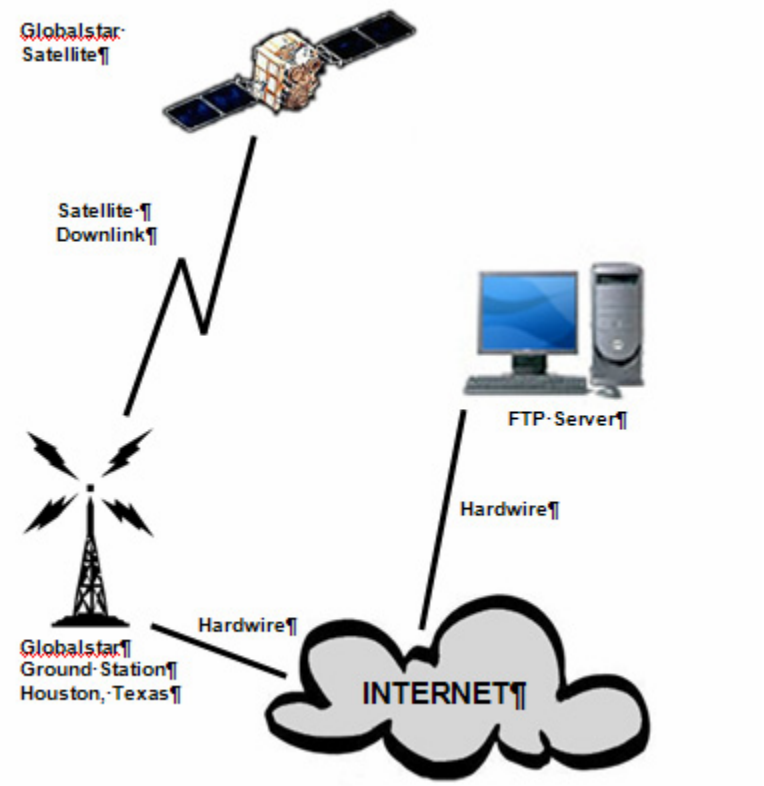



Figure 48. TSSRv3 downlink(TNT report, 2005)

## 4. Software Components

To utilize the hardware and network components, the TSSRv3 uses software functionality written in C Sharp (C#). The software part of the system serves the different

system's requirements. it is divided, based on them, into six different modules: the Capability, Acquire, Loadshare, Master, Loadshare Slave, and Upload.

The system's information flow begins with the Acquire Module, which controls the Creative WebCam and capture the digital image. Then the Capabilities Module, responsible for discovering the device's capabilities, senses the new image and writes it in the proper text file on the device. Next, the Loadshare Master in the central node, which controls all the other nodes, receives the new capabilities text file. By using the loadshare algorithm, it determines if the node will transmit the image or whether another node will take the transmission responsibility. The algorithm tries to keep the network's balance. Finally, the Upload Module takes control and transmits the image to the FTP server. The following figure completes the above software's functionality overview.

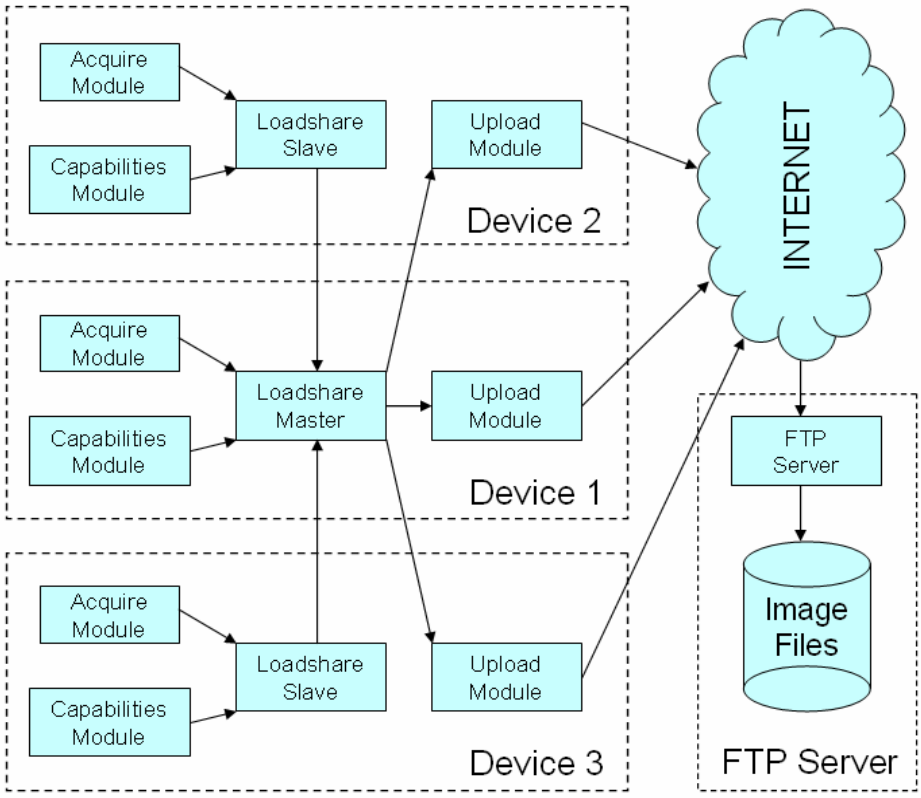



Figure 49. TSSRv3 Software Modules and Information Flow (TNT report, 2005)

# IV. OBJECT-TRACKING APPLICATION: ARCHITECTURE AND IMPLEMENTATION

This chapter presents the object-tracking application, the main objective of the thesis. The application is an algorithmic-software implementation written in Java, which is capable of receiving Active Messages (AM) and generating object detection alerts and motion estimation data. The implementation of the object-tracking application makes use of the Crossbow sensor motes and uses the background knowledge introduced in the two preceding chapters. The motivation for the object-tracking application is to demonstrate a complete and useful application sited in the systems control station that uses a general purpose, commercial-of-the-shelf (COTS) sensor network system. The application acts as an interface which collects the raw data (WSN's messages) that the motes generate (numerical values). It makes the proper comparisons and algorithmic processes to identify and track the object while it is moving inside the area covered by the wireless sensor network. Finally, with the detection and tracking data, it alerts the user or another system. The following sections provide an overview of the object-tracking application.

## A. APPLICATION REQUIREMENTS AND DESIGN CONSIDERATIONS

There are a number of critical factors that influence the design and the implementation decisions for the object-tracking application. The most important design consideration is the purpose and the deployment environment of the application. Based on the sensors' characteristics, the application aims to support detection and tracking of different kinds of objects in different deployment environments. The application supports the detection and tracking of humans and vehicles. These kinds of objects can be detected and tracked when they are moving in a corridor or on a road. Thus, the system supports indoor and outdoor deployments.

Another important consideration is the wireless sensor network system that the object-tracking application uses. The Crossbow MSP410 Mote Security System, discussed in the previous chapter, is the sensor network system that serves the data collection for the application. The MSP410 system is the application's interface with the environment; its characteristics are the tools feeding the application with the sensor's returns, but they also impose limitations. The MSP410 provides detections based on the

returns from the passive infra red (PIR) and magnetic sensors. Additionally, it supports ad-hoc mesh networking and data forwarding to the system's gateway, which is directly connected to the application workstation. Moreover, the motes are battery-powered; the gateway is not. These are some of the important characteristics of the MSP410 that they have been discussed in the chapter 2.

The application must maintain a balance between the cost of deployment and reliability. To achieve this, the implementation tries to minimize the number of required WSN motes and at the same time, it tries to provide the highest system reliability. This could be done by the optimum usage of the MSP 410 characteristics. Proper topology selection and the appropriate communication and sensing deployment ranges are some of the design decisions that must be considered.

A key design decision related to the application development is that all the algorithmic manipulation and decision-making work is performed in the system's base station by the object-tracking application. The MSP410 does not do any kind of data processing, selection, or filtering; it just collects and forwards all the data. This factor was derived from the following considerations. First, this approach is the most straightforward and simple: it keeps the system simple, and makes the necessary computations at the ends. In addition, the expected number of WSN nodes that the system expects to use is small. Thus the overhead that is produced by the data forwarding to the base station, without any in-system filtering or manipulation, is estimated to be low. Finally, the MSP410 system does not support the reprogramming of the nodes' software package.

An additional function of the object-tracking application is to provide interfaces to the user and other systems. The graphical user interface should be simple, providing the user with the ability to properly configure the system and also inform him/her of the application output. In addition, the interface to other systems should be kept as simple as possible.

Finally, the application design involves considerations about the logic of the object-tracking algorithm. The focus is to keep the logic, the algorithm, and the software implementation simple. During the sequential development of the application, the

designer can adjust the algorithmic implementation to meet new requirements and overcome possible difficulties. Moreover, the design's simplicity helps in the apprehension of the algorithm and software implementation. The following sections further analyze the application requirements.

## B. APPLICATION SCENARIOS

There are several deployment scenarios that the developer can choose from. The choices depend on the different systems' parameters and configurations. The area of interest that the application has to cover, the kind of object that the application has to detect and track, the nodes' sensing and communication characteristics, the available number of nodes, and the WSN architecture are some of the factors that affect the deployment scenario selection. Most of those factors have been analyzed in the preceding chapters.

For the object-tacking application, the designer's choice of deployment scenarios is based on the following. The scenarios have to cover both indoor and outdoor deployments. The area of interest is specific and narrow. The application aims to detect and track objects that are moving along a road or a corridor. Scenario selection is also affected by the nodes' characteristics. The MSP 410 nodes are able to identify and detect objects based on their PIR and magnetic returns. Thus, the sensing ability of the nodes specifies the kind of objects, humans and vehicles that can be detected and tracked. Moreover, the available number of MSP 410 nodes that the object-tracking application has available is specific, with a maximum of eight. The WSN architecture is another factor that affects the scenario selection. The MSP 410 WSN kit is able to support only flat network architecture. Taking also into account the available number of nodes for deployment, the flat network architecture is adequate. Finally, the deployment scenarios have to be simple and, at the same time, general. The scenario's simplicity is important for its easier understanding and use by the application's user. It must also be general, in order to cover most of the deployment environment (road and corridor design and conditions).

Predetermined deployment (chapter 2) is the strategy used for the object-tracking application. As the deployment environment is known and the number of nodes small, this kind of strategy provides the ability to control the area of coverage and maintain the

high QoS. There are three predetermined deployment scenarios: the straight-road scenario, the T junction, and the crossroads scenario. Although only three, they are suitable for most of the possible deployments.

### 1. Straight Road Scenario

The straight-road scenario (Figure 50) is the simplest. It covers all the road and corridor designs with and without curves and dips. The assumption is that, during the deployment, the nodes maintain their RF connectivity. One constraint in this scenario is that for a given part of the road or corridor that the system monitors, the road or corridor must not have any exits. Any object that enters the system from one side has to exit from the other side.

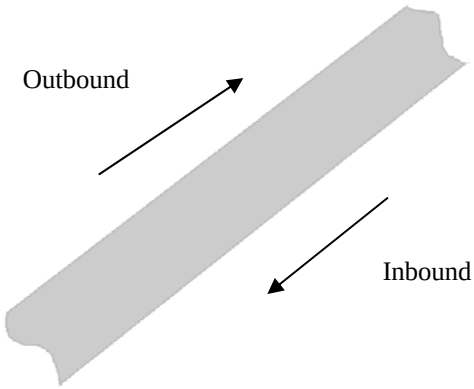



Figure 50. Straight-Road Scenario and its main directions.

### 2. T-Road Scenario

The T-road scenario (Figure 51) is an extension of the straight-road scenario. It is a combination of two straight roads or corridors, in which one road or corridor ends in another. Any object at the end of the first road must turn left or right. This scenario is more complicated than the straight-road scenario, in that it must produce the direction outputs. In addition to the “inbound” and “outbound” directions that the straight-road scenario supports, the T-road scenario supports a “left to right” and “right to left” direction and the combination of those four directions.

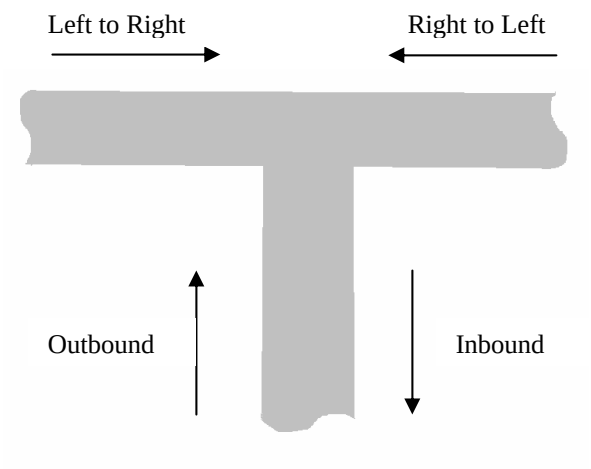



Figure 51. T-Road Scenario and its main directions

### 3. Crossroads Scenario

The third and last scenario is the crossroads scenario. It is based on the above two and is the most complicated in its implementation. This scenario supports four main directions and their combination. The main directions are: inbound, outbound, left to right, right to left, and north to south, south to north. Figure 52 presents the crossroad scenario and the main directions it supports.

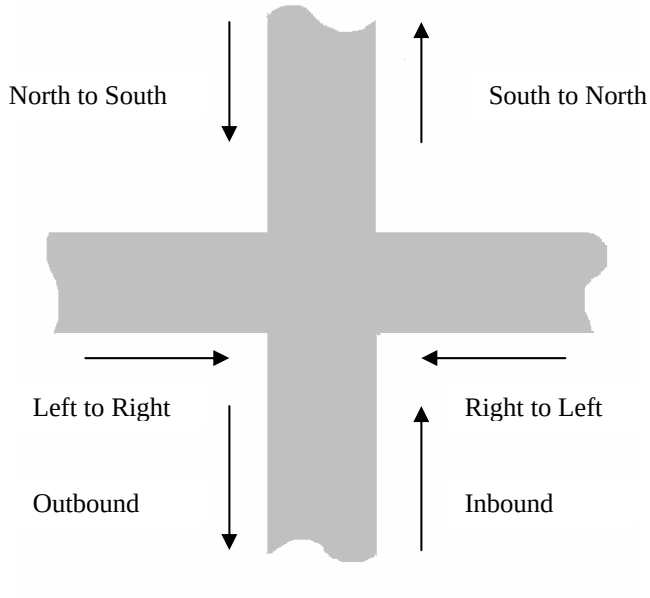



Figure 52. Crossroads Scenario and its main directions

## C. FINDING SENSOR'S TOPOLOGY

There are several ways that the nodes of a WSN can be deployed. As was mentioned in the preceding section and in chapter 2, the most important factors that determine the nodes topology are: the node characteristics, the network architecture, the deployment environment, and the application requirements. Other important factors are the reliability and redundancy that the application has to support. For the object-tracking application, based on the application requirements and after the first experimental results, we choose a high-level of reliability and redundancy.

The proposed Crossbow deployment techniques (as described in chapter 2) are very general and are not adequate for the current application. However, the scenario selection discussed above bound the node deployment topology. Thus, given the deployment scenarios and the nodes' characteristics the nodes have to be deployed along the road or corridor. The distance between the nodes is determined by the communication and sensing ranges, the reliability and redundancy level, and the application latency.

In light of the MSP 410 experimental results we determined that the maximum deployment distance between the nodes for the object-tracking application is from 45 to 65 meters. The sensing range and the horizontal field of view for each node determine the nodes' deployment. The horizontal field of view for each of the four PIR sensors of a MSP 410 node is 45 degrees; the two magnetic sensors provide 360 degree horizontal coverage. The sensing range, especially for the PIR sensor, is also affected by environmental conditions, especially temperature. The MSP 410 experimental results indicate that the sensing range for both PIR and magnetic sensors is at least five meters from the node. For simplicity and accuracy purposes, we do not want any overlapping of the nodes' sensing areas. To summarize, the nodes must maintain proper distances between them in order to communicate, but the distances have to be enough to avoid the confusion produced by duplicate detections.

The connectivity redundancy level that we chose for the object-tracking application is two. That means that each node is able to communicate with two neighbor nodes on each side. The level-two redundancy enables the application to continue working even if it loses a number of nodes. The final issue that determines the node

topology is the space that the application has available for deployment, the width and length of the road or corridor. The following sections analyze the system topology for each scenario. For the deployment we assume that we have eight nodes and one base station available.

The final factor that affects the nodes topology is the system delay. The total delay is the sum of three different delays. The first system delay is the time that the node needs to detect the object and produce the detection signal. The second delay is caused by for the transmission of the message from the source to the gateway and base station. The final delay is produced in the base station by the algorithmic manipulation of the data and the input-output procedure. Although we do not have experimental results for total-system delays, we assume that they exist and we use "sufficient" distance between the nodes.

### 1. Straight-Road Node Topology

For the straight-road scenario, the nodes are deployed along the road. They can all be positioned on the same side or both sides of the road/corridor. Although we normally deploy the nodes by using both sides of the road, the deployment pattern mainly depends on the width and construction of the road. For a narrow road we can chose to deploy the system on only one side of the road. For a wide road the two-side deployment is preferred. Figure 53 illustrates a deployment where nodes are placed on both sides of the road. The normal distance between the nodes that we used is 20 meters, but it can be adjusted based on each deployment's specific needs. Ideally, the sensing areas of two neighbor nodes are tangential. The width of the road and the size of the sensing area also affect the accuracy of the detection, especially the PIR detection. In a wide road where the object passes away from the nodes, the detection is less accurate than in a narrow road. This happens because of the way the PIR sensor detects movement (chapter 3). Finally for the straight-road deployment, we used only one of the MSP 410 node's four PIR sensors (quad 1).

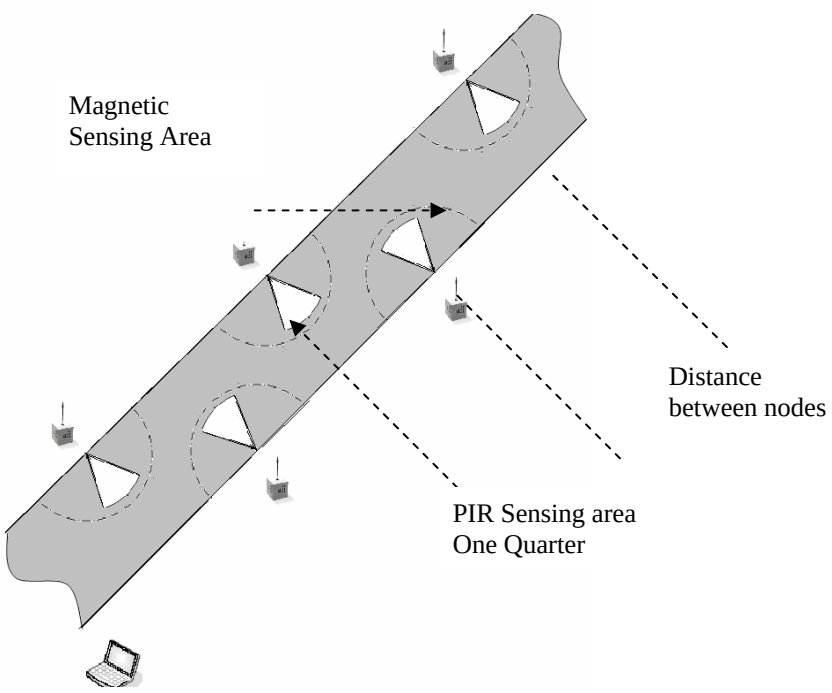



Figure 53. Node topology in the straight-road scenario and the magnetic and PIR sensing area

### 2. T-Road and Crossroads Node Topology

The T-Road and crossroads scenarios have the same principles and considerations as in the straight-road scenario. The main difference is that, for the nodes positioned is at the corners of the roads, we use two quarters (two PIR sensors) for the PIR detection, instead of one. By taking advantage of the nodes' characteristics, we eliminate a number of important nodes for deployment. Figure 54 presents the T-road scenario while figure 55 shows the crossroad scenario.

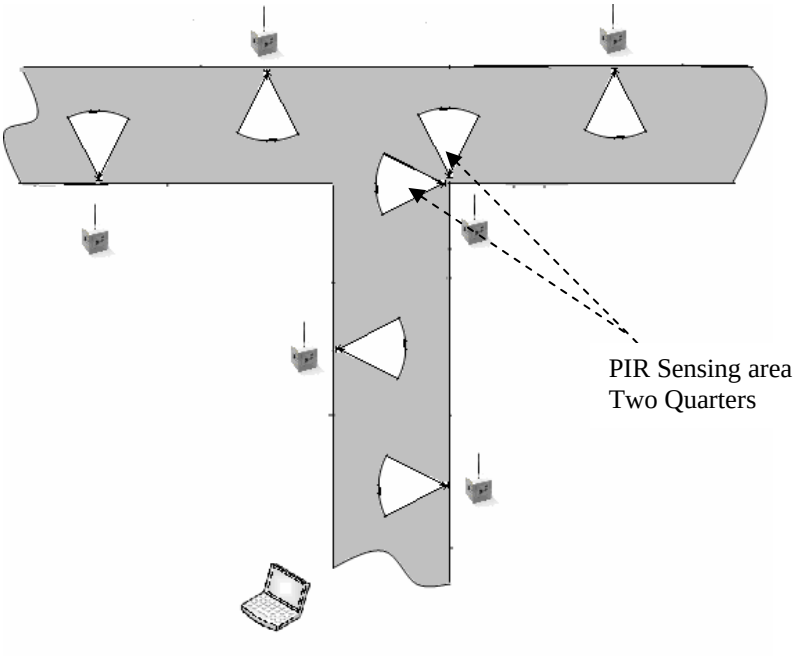



Figure 54. Nodes topology in the T-road scenario and the PIR sensing area.

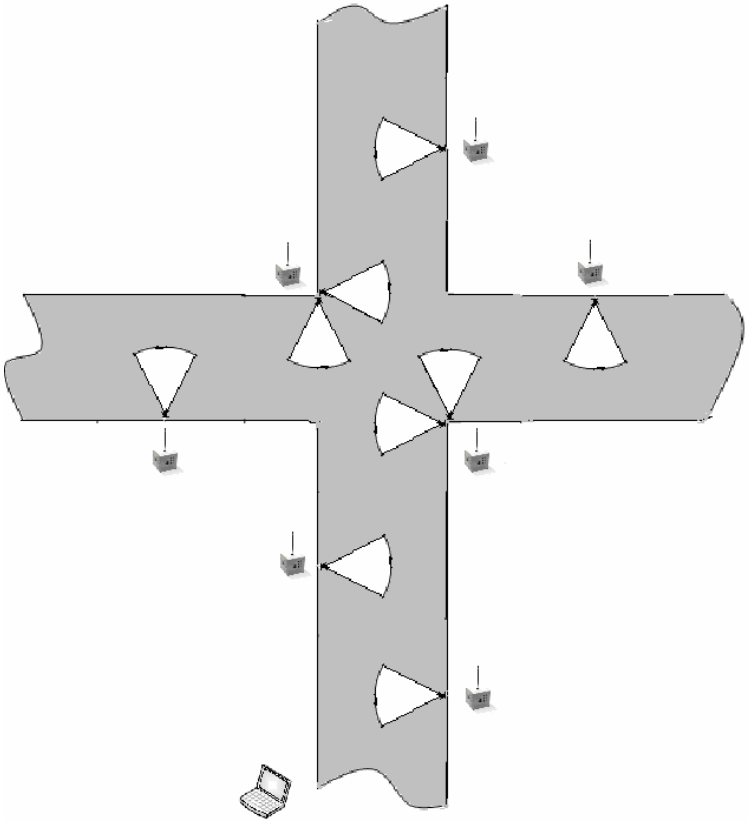


Figure 55. Nodes topology in the crossroads scenario and the PIR sensing area.

## D. PROGRAMMING LANGUAGE

We chose Java as our main programming language. There were a number of factors that influenced our language selection. One critical consideration is that Java is a widespread language with a great variety of libraries and available code. Java's networking libraries are especially helpful. In addition, the application's implementation in Java supports its interconnectivity with other possible Java projects.

Another important factor that influenced our language selection is the object-tracking algorithm itself, which will be analyzed later. The object-tracking algorithm is a simple long series of selections that checks and moves detection objects.

## E. OBTAINING DATA FROM THE SENSOR NETWORK

The next important step in the development of the object-tracking application is to obtain the data that the MSP 410 wireless sensor network system captures and forwards to the base station. The route that the data follows is from the MSP 410 node directly, or through neighbor nodes, to the MBR410CA, the WSN's gateway. From there, the data is forwarded through a serial port to the computer that runs the object-tracking application. When the data arrives at the serial port, the application must capture it.

The MSP 410 uses the TinyOS Active Messages (chapter 2) format to transmit the data. The wireless sensor network captures the information and encapsulates the data inside the AM, which then it is forwarded toward the system's gateway. To forward the data to the application's base station, the gateway forwards the AM using the RS 232 protocol. The application listens to the serial port, it captures and reconstructs the AM, and by removing the headers, it obtains the actual information. As shown in Figure 56, the default length of the AM is 36 bytes, as described in the University of California AM.h v 1.4 header file of the (2000-2003). Since the AM length is not restricted, for the MSP 410 system Crossbow uses a length of 38 bytes.

| 2 Bytes | 1 Byte | 1 Byte | 1 Byte | 29 Bytes | 2 Bytes |
|---|---|---|---|---|---|
| **Address** | **Type** | **Group** | **Length** | **Data** | **CRC** |

Figure 56. AM message format, University of California (2000-2003)

The object-tracking application's Java component-class responsible for obtaining the data is the SimpleRead class (it is further analyzed in the software component section ??). This class is responsible for receiving the AM from the serial port and extracting the useful data. Then it forwards the data to the rest of the application components.

The data field format in the AM is not specific; it is up to the developer to use all the available size or only a part of it. The developer can place the data in any order he/she wants in the data field. During the SimpleRead development, we had great difficulty in determining the useful data bytes inside the AM data field. Although we had available a similar source file from the Crossbow (Crossbow msp410.c,v 1.4, 2004; Crossbow xlisten.c, v 1.17, 2004; Crossbow xsensors.h, v 1.35, 2004) and oral instructions, we did not have available the actual Crossbow data field format. Therefore, we spent a great amount of time identifying it. The final message format that we concluded with, which contains the important information for the object-tracking application is presented in figure 57.

| **10 B** | **1 B** | **7 B** | **1 B** | **2 B** | **1 B** | **1 B** | **2 B** | **2 B** | **2 B** | **9 B** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Headers & networking data** | **nodeid** | **Unidentified bytes & multihop header** | **parentid** | **Seq#** | **vref** | **quad** | **pir** | **mag** | **audio** | **Unidentified bytes & trailers.** |

Figure 57. Data-field message format for the MSP 410 system.

The object-tracking application uses most of the information that is included in AM. The nodeid identifies the MSP 410 node that sends the message; the seq# is a unique sequence-number value for each message per node. The vref value identifies the voltage value related to the node that sends the message. The quad and pir values are related to the passive infrared sensor. The mag value is returned when a node has a magnetic detection. The audio field is currently unavailable in the MSP 410 system. In accordance with the msp410.c, v 1.4 (Crossbow, 2004), after the audio value they follow the pirThreshold, magThreshold, and audioThreshold reference values. The object-tracking application uses the nodeid to identify the node that sends the message, the seq# to separate each message, and as utility values, the quad, pir, and mag values to identify

the detections and, based on them, to produce outputs related to the object's direction and speed. Finally, the vref and the parentid are only displayed by the application as an indication of the node's battery and the network condition. Whenever the SimpleRead component receives a message, it forwards it to the other application's components. If the object-tracking application then detects an object, it encapsulates the data into a "target" object and implements the analysis that is described in the next section. More information about the MSP 410 system is included in chapter 3.

## F. ANALYSIS OF RAW DATA

The algorithmic analysis of the data is the core of the object-tracking application. The logic that the algorithm follows is simple. Whenever a node detects an object, it checks whether it has any information about the object. Then it creates or updates the tracking characteristics of the object and informs the neighbor nodes about the object currently in the system. The application also informs the user and any other applications active at the time. The algorithm can be divided into two parts and follows several steps, as described below.

The first part is responsible for object detection. The application receives messages from the MSP 410 WSN system and for each message it determines whether it is related to object detection or whether it is a message that facilitates networking such as routing table updates. The following sections describe the algorithmic steps of the first part.

### 1. Step 1: Object Detection

To identify an object detection message, the application performs a series of comparisons. It compares the received PIR and mag value with their corresponding thresholds. The thresholds reflect the current deployment configuration. Whenever a received PIR or magnetic value is above the threshold value, the application assumes that object detection has occurred. The quad value is also related to the PIR returns. Its value represents the MSP 410 node's quarter that has detected a change at the PIR value; otherwise its value is zero. Because the application works with specific scenarios, and the nodes have specific predetermined topology for each scenario, the application uses the quad value as an additional selection criterion to determine when PIR detection happens.

### 2. Step 2: Characterization of the Detected Object

The second step is responsible identifying the kind of object that it detects. It does that by evaluating the source that produced the detection. If the object detection is based only on the PIR value, the application concludes that the object is not a vehicle, and it is probably a human. If the object detection is based also or only on the magnetic returns then the object probably is a vehicle.

### 3. Step 3: Storing Object Data

Subsequently the algorithm must store the data. Every time the object-tracking application identifies a detection, it creates a "target" object that holds all the related with the detection data. In addition to the variables that hold the detection data, the target object uses others to hold information related to the motion of the object (e.g., identification, direction, speed). The second part of the algorithm is responsible for producing this information. The object-tracking application initially creates "application node" objects, which correspond to the physical nodes of the MSP 410 WSN and are used to hold data related to them. In addition, the application uses a first-in first-out (FIFO) data structure for each of the application nodes (the number of application nodes is the same with the number of MSP 410 physical nodes). The FIFO data structures are used to store the target objects that concern each application node.

### 4. Step 4: Updating the Thresholds

The final step of the first part is responsible to keep update the application's thresholds. The PIR and mag thresholds are constantly updated based on the messages received. Whenever an AM from the WSN is not characterized as an detection message it is used from this part of the algorithm. Although the primary reason for those messages is networking, they return values for all the message fields (e.g. PIR, mag, nodeid). These PIR and mag returns, demonstrate the environmental conditions in the deployment environment. The application's methods responsible for the thresholds updates, capture these PIR and mag values and updates the thresholds.

The second part of the object-tracking application is responsible to produce the outputs. It receives the target objects from part. It has also available the information stored in the "application node" objects and in the nodes' FIFO data structures. Based on the above information the algorithmic manipulation continues with the following steps

### 5. Step 5: Checking the node FIFO

Every new target object that is passed from the first part to the second, by default, in the direction and speed variables, contains the “unknown” and “zero” values respectively. Every time that an application node produces a target object (the corresponding MSP 410 node returns an object detection), it checks the corresponding FIFO. If the FIFO is empty, the application node does not change the values of the direction and speed variables (they remain “unknown” and “zero”).

If the FIFO is not empty it can contain two different kinds of target objects: those that have “unknown” and “zero” in the direction and speed variables, and those that have a specific direction (based on the scenario), and speed values produced earlier by a neighboring application node. In the case that FIFO contains target objects the algorithm continues by producing motion outputs.

### 6. Step 6: Producing the Direction Output

This step is responsible to produce the direction outputs. As it is mentioned above if the FIFO is empty the algorithm is not able to calculate the direction of the object and keeps the default value (unknown) that the target object initially contains. In the case that the FIFO is not empty and the stored direction and speed data, in the first stored target object in the FIFO, are the default, the application node checks the ID of the application node that stored the data in its FIFO. Then based on the nodes’ deployment topology and the stored and current node, it calculates the direction. Finally, the produced direction value is stored in the current target object.

Assuming that the first target object in the FIFO contains direction and speed which are different from the unknown and zero, the application node performs the same step described above and updates the direction variable in the current target object.

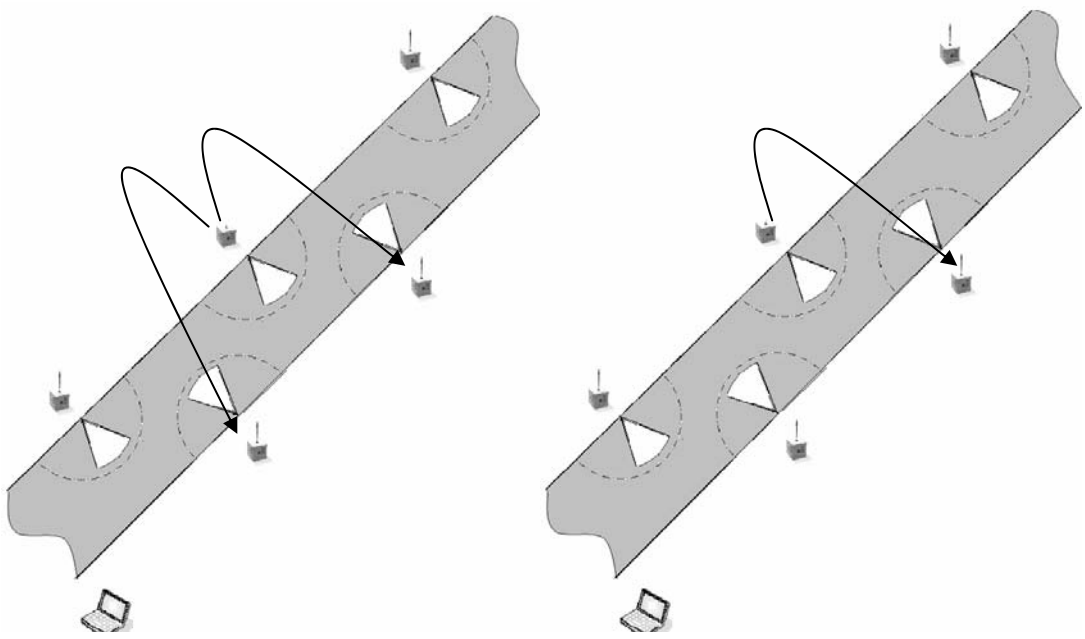
### 7. Step 7: Producing the Speed Outputs

If the FIFO is empty in this step the algorithm also does not change the default speed value of the current object. Otherwise, by checking the time difference between the time that the stored detection happens and the current time and using the range between the nodes, it produces the speed for the tracking object. The object’s speed is also stored in the current target object. In addition, this step is responsible to keep the object’s speed history. For that purpose an additional data structure is used as a variable of the target

object. Thus, for a new detection the speed history data structure is initialized and is updated with the produced speed values as the object is moving through the system. The speed history data structure contains information about the node id that had the detection and the corresponding object's speed.

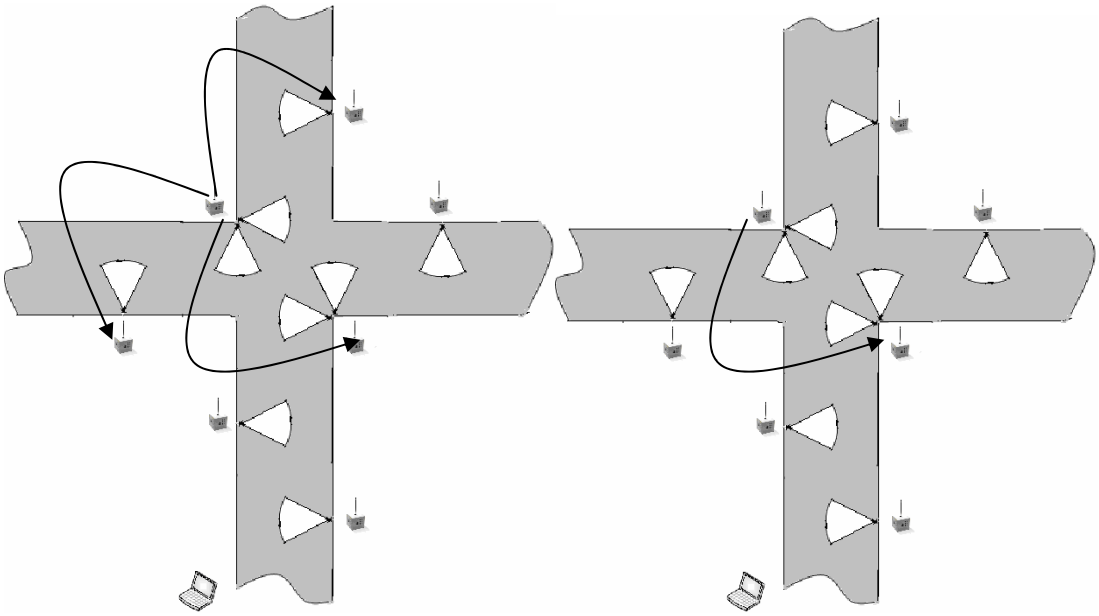
### 8. Step 8: Informing the Neighboring Nodes

After the completion of the above steps the algorithm "informs" the neighbor nodes about the detected object. The application "informs" the neighboring nodes by storing the updated target object in their FIFO. The algorithm selects, based on the object's direction and the deployment scenario, which nodes must inform. If the direction is unknown informs all the neighboring nodes. In the T-road and crossroads scenarios, the "corner" nodes also inform themselves. That happens because the corner nodes use two PIR sensors instead of one as is the case with the other nodes. Finally, if the direction is specific, the algorithm informs only the nodes in the given direction (Figure 58 & Figure 59).



(a) Target Object with unknown direction (b) Target Object with known direction (outbound)

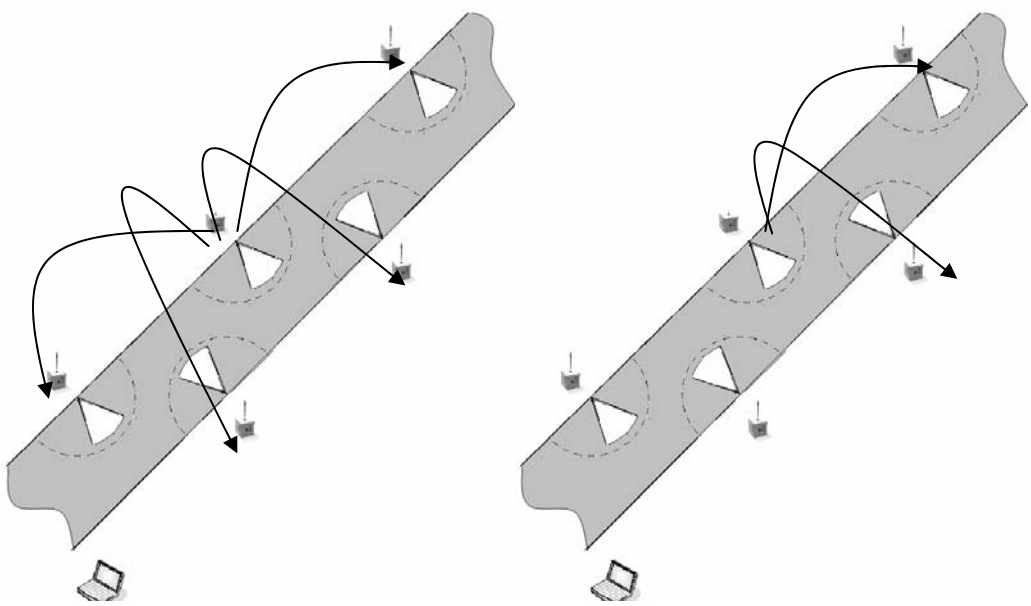
Figure 58. Neighbor Nodes' updates for a target object with and without knowing the object's direction for the straight road scenario



(a) Target Object with Unknown direction (b) Target Object with known direction (left to right)

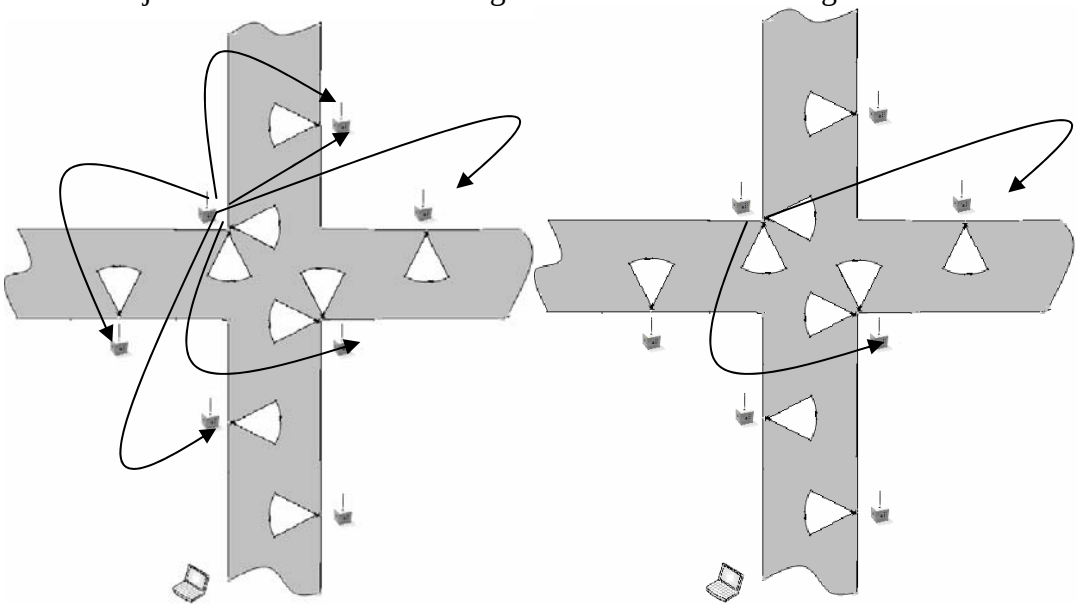
Figure 59. Neighbor Nodes' updates for a target object with and without knowing the object's direction for the crossroads scenario's corner nodes

In the first approach of the object-tracking algorithm that we implemented, every time the application node had an object detection, it informed only the immediate next neighboring nodes (Figure 58 & Figure 59). In the new version, instead of only the immediate neighbor, the application node informs two nodes in the direction of approach of the object, as Figure 60 and Figure 61 illustrate. This change increases the system redundancy and the programming complexity but at the same time it improves the system reliability as now the application is able to produce tracking outputs even if a node loses a detection or a node is out of order.



(a) Target Object with Unknown direction (b) Target Object with known direction (outbound)

Figure 60. Neighboring Nodes' updates for a target object with and without knowing the object's direction for the straight road scenario in the algorithm's second version



(a) Target Object with Unknown direction (b) Target Object with known direction (left to right)

Figure 61. Neighboring Nodes' update for a target object with and without knowing the object's direction for the crossroads scenario's corner nodes in the algorithm's second version

### 9. Step 9: Removing the Old Data

The final step that the algorithm performs is to remove the old data that FIFOs contains. Whenever the application node checks its FIFO, finds that the queue is not empty, and compares the stored data with the current target object, when the comparison is finished, it removes the old target object from all the application's FIFOs that have it. This happens because the stored information has been used and is no longer required ; thus, it has to be removed to make space for the next detection and to avoid confusion.

## G. PROGRAMMING ISSUES AND ASSUMPTIONS

Although the scenario and the algorithm that the above sections describe are quite simple, their programming implementation is quite complex. Given the MSP410 characteristics and the design choices mentioned above, the programming implementation uses the following three assumptions to accurately produce the required data outputs. The object-tracking application results are based on the information that the MSP 410 wireless sensor network provides. Thus, the results' accuracy and the detection probability of the object-tracking application are based on the WSN. Second, the application assumes that inside the system there is only one object. The concept of a wireless sensor networks is not the best proposal for a heavily trafficked road. The third assumption is that the WSN is specific. The number of the available MSP 410 nodes is eight.

Taking into consideration that the object-tracking application is used by a battery-powered system, we tried to minimize the run-time overhead.. First, we implemented the application as an event-driven program. Whenever the WSN returns a message through the serial port, it captures it and triggers and forwards the useful information to the rest of the application.

In addition, we tried to keep the memory and CPU requirements as low as possible. The program uses objects for the application nodes, which are used to hold only the important data related to the physical characteristics of the nodes and their position. The number of the application node's objects is related to the number of the physical nodes avoiding the unnecessary memory usage. The Java code also uses "target" objects

to hold and transfer the information related to the detection events among the program's

components. After the application uses a target object, it removes it and frees memory space.

Another programming issue was the simplicity of the algorithm's implementation in Java. We tried to implement the algorithm, in with a straightforward way and, at the same time, keep the code in a reasonable size. Like a lot of the algorithmic implementation, the object-tracking application contains sequences of selection statements. Those selections check the incoming data with the related thresholds and the older data, in order to produce outputs about the object's movement. In the Java code we tried to keep the code length to a reasonable size to avoid redundant statements without sacrificing the code's simplicity.

The final issues were about the application interconnectivity and the user interface. The implementation effort for object-tracking application as a new wireless sensor application is in the algorithmic process of the sensor network's data. Thus, the interconnectivity and user-interface features are at a primitive stage. Currently, the application only feeds the user with outputs only by displaying the object's direction, speed, and speed history and the TSSRv3 system, which triggers in order to take and transit pictures whenever the object is in a proper place. The interconnection of the object-tracking application and the TSSrv3 system is implemented by using a simple input/output trigger (the object-tracking application renames a txt file into the C:/loadshare directory; the TSSRv3 senses the change, takes the picture, and renames again the text file). Finally, the current version of the application supports the user with a simple configuration, control, and display interface. The following section provides an overview of the tracking object Java components that implement the aforementioned scenarios, algorithm, and programming issues. (The actual code is included in the Appendix A.)

## H. SOFTWARE COMPONENTS

The software used to implement the object-tracking application and satisfy the application's requirements is a Java package of three main components. The eight different Java classes that the package uses can also describe the software implementation. All these modules serve different functions, but they work together to achieve the application requirements set for the object-tracking application. The three

components as shown in Figure 62 are: the user interface, the data acquisition, and the algorithmic process. The component separation is used mostly for documentation purposes, because the boundaries between the components are very loose.

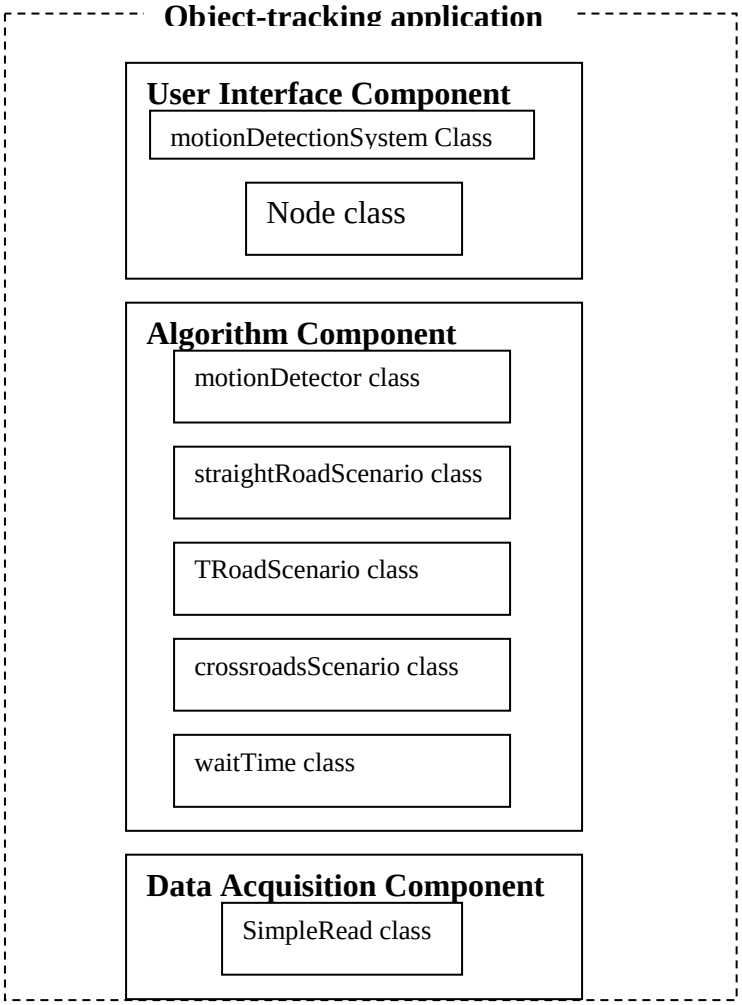



Figure 62. Object-tracking application components

In addition to the three main components, Figure 62 presents the Java classes that construct each component. The application classes are: SerialReader, node, waitTime, motionDetector, straightRoadScenario, TRoadScenario, crossroadsScenario, and the objectTrackingApplication. Figure 63 and Figure 64 illustrate the UML diagram for the object-tracking application. Figure 63 also includes the inner and nonpublic classes not mentioned above. Given the component separation, the following sections present the purpose and the general functionality that the above classes serve.

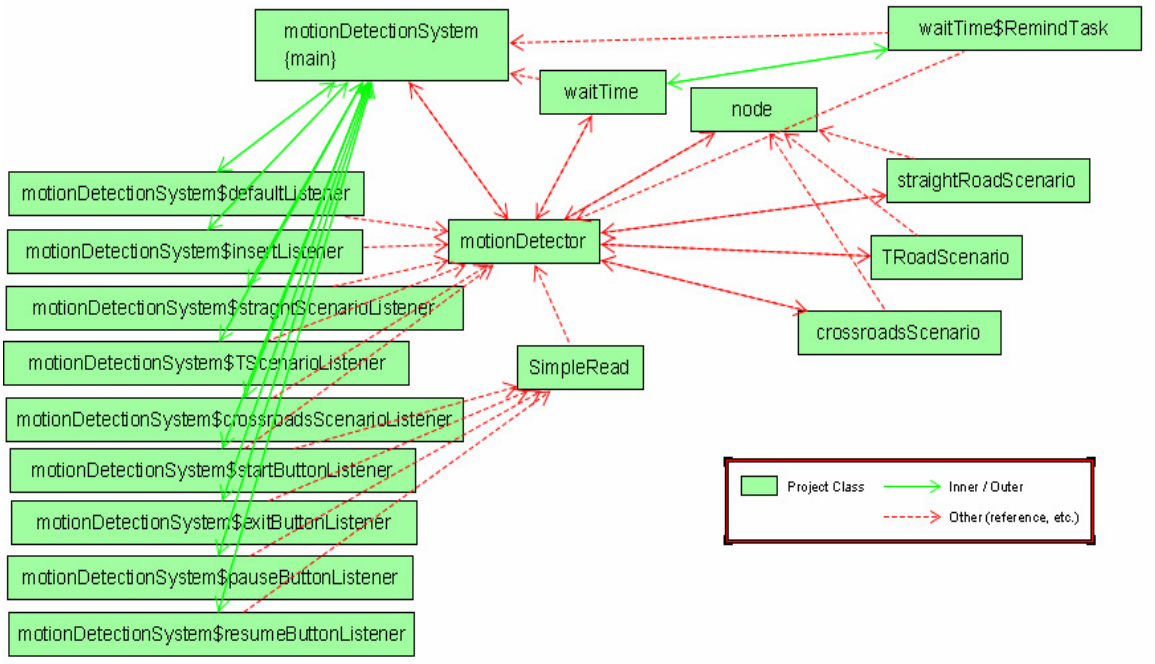



Figure 63. Object-tracking application: Complete UML diagram

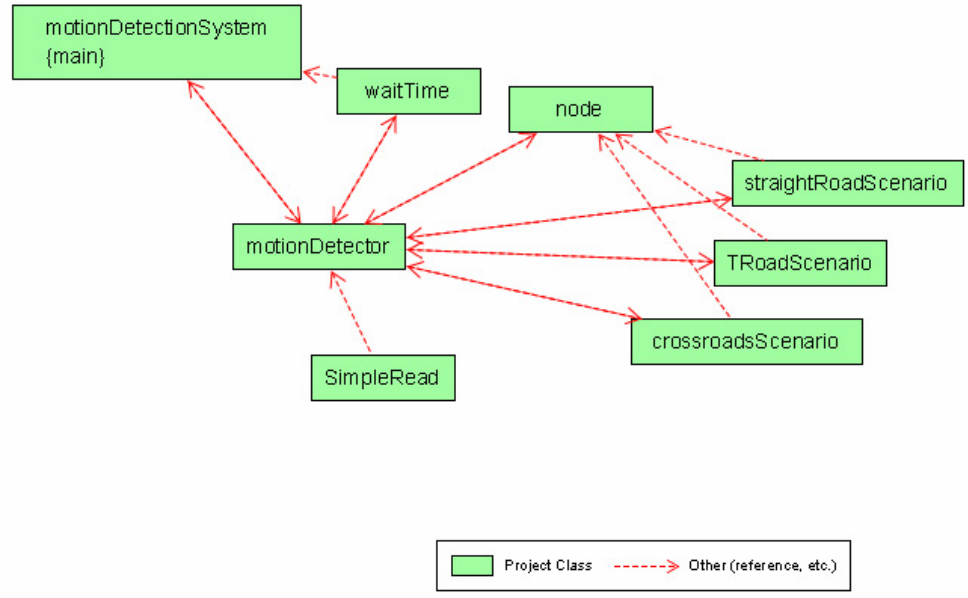



Figure 64. Object-tracking application: Simple UML diagram

### 1. User Interface Component

The user interface component is important for the object-tracking application as it supports of a variety of functions. This component has three different purposes: it helps

the application's deployment configuration, it provides operational control, and it is used to display the application's outputs. Figure 66 provides an overview of the object-tracking application GUI window.

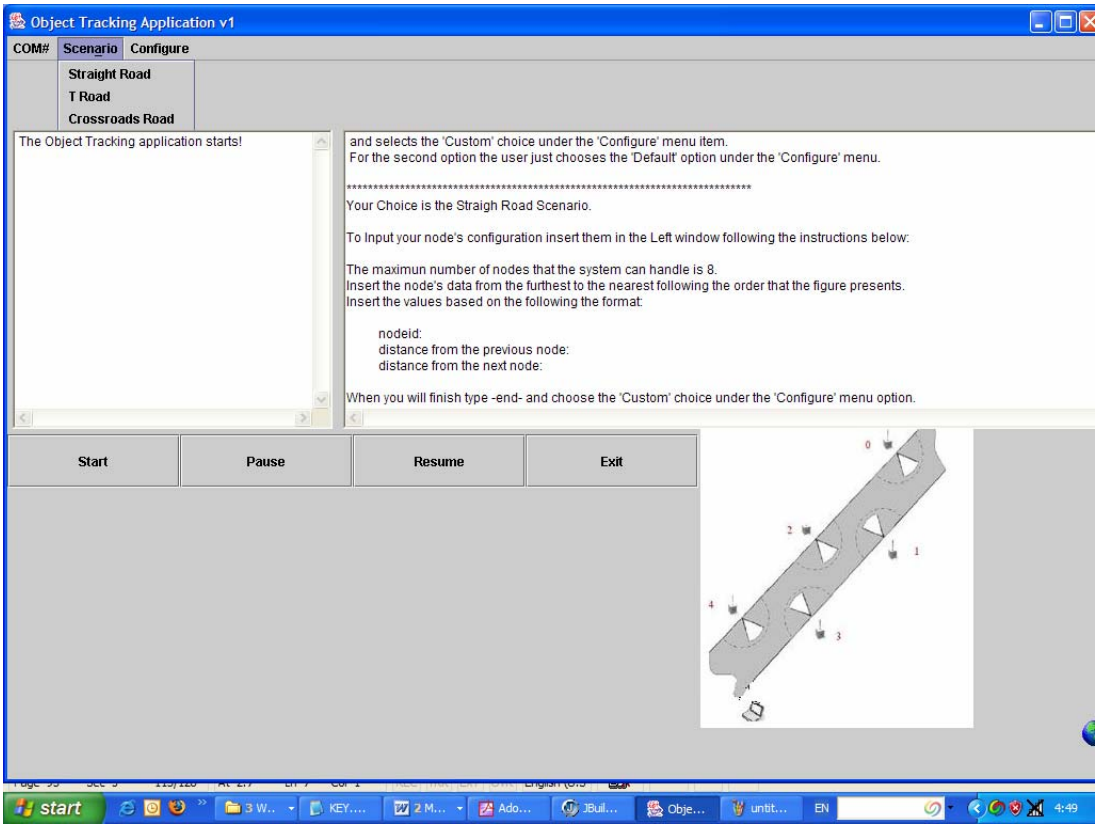



Figure 65. Object-tracking application's user interface overview

When the object-tracking application starts, the right window displays the initial instructions for the user. The configuration starts when the user selects the desired deployment scenario from the three choices under the "scenario" menu button. The provided choices correspond to the three deployment scenarios: straight, T, and crossroads. The next step is the nodes configuration. The application supports two different node topological-configuration choices, the "default" and the "insert". If the user decides to configure the deployment it chooses the "insert" menu choice. Instructions about the custom node configuration are provided after the scenario selection, when a text document and a figure appear, in the upper-left and bottom-right corners of the applications window respectively, appear providing instruction about the

nodes' configuration. They provide the text format that the user must use to insert the nodes' physical characteristics (nodes' id and distances between the neighbor nodes), and the exact order to follow to insert the data. The default configuration is recommended only for demonstration purposes. It uses the figure's node configuration and constant distances between the nodes.

The GUI window also contains a set of control buttons. When the configuration step is finished, the user is able to start the data acquisition and the algorithmic components by pressing the "start" button. Later the user can also use the "pause," "resume," and "exit" buttons as desired to control the application.

Finally, the user interface component is used to display the application output. The left window is used to display the commands that the application sends to the TRSS system. The right window is used to display the direction, speed, and speed-history outputs that the application produces.

The motionDetectionsystem class and the node class are the two Java source files that the user interface component use. Figure 66 and Figure 67 present the class diagram of the motionDetectionsystem class. The motionDetectionsystem class contains a variety of inner classes and methods to support the GUI functionality. In addition, the motionDetectionsystem class contains the "main" method of the application responsible for initiating it. In addition to the above tasks, this class also is responsible for initiating the proper data structures that the rest of the program uses.

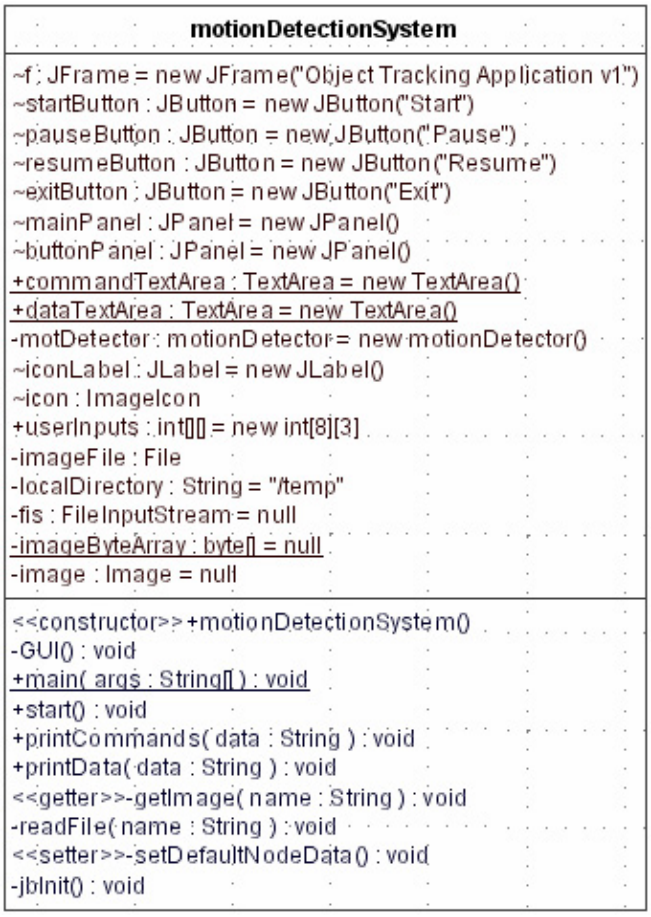



Figure 66. motionDetectionsystem class diagram

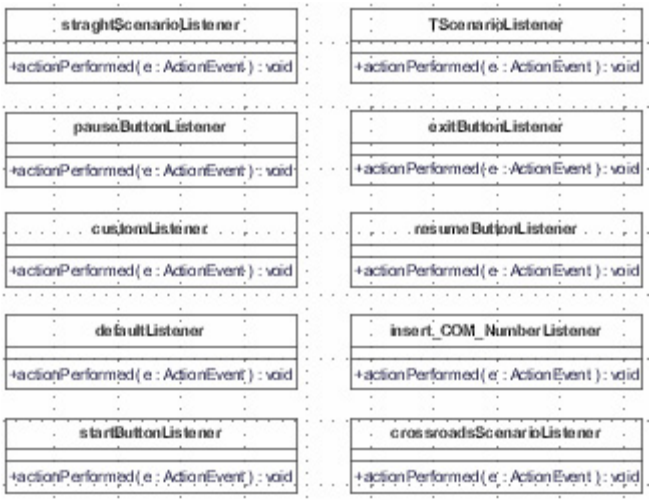



Figure 67. Class diagram for the motionDetectionSystem inner classes

The node class presented in Figure 68 is also contained in the user interface component. We consider the node class as a utility class that serves storage purposes. The node configuration data that the user inserts, or the default, is stored in the node class objects to be accessible from the rest of the application's classes.

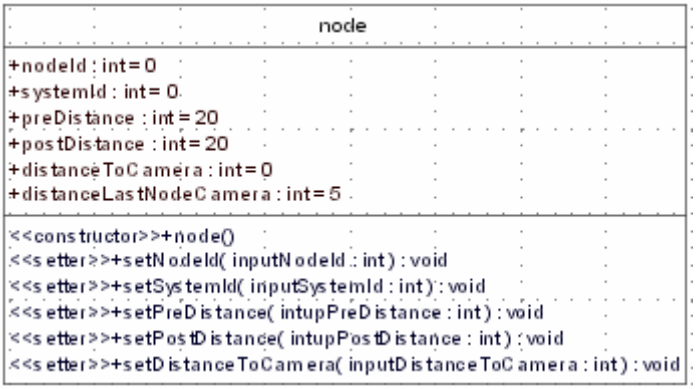



Figure 68. Node class diagram

### 2. Data Acquisition Component

The data acquisition component contains the SerialReader class. This class provides the proper functionality for the data acquisition. First, it creates, opens, and maintains a serial port connection (RS 232). When the connection is established, the class is ready to receive the data returned from the MSP 410 system through the serial port. Whenever the WSN returns an AM, the SimpleRead class receives it. Then by knowing the AM and the data field format, it extracts the raw data that the sensors return. It stores the message's data in an array that passes to the algorithm's component to qualify it. It also is responsible for displaying the raw values that it receives from the WSN in a command line window. Finally, the data acquisition component operation is controlled from the user interface component through the functional buttons that the motionDetectioSystem GUI provides. Figure 69 shows the class diagram for the SerialReader class.

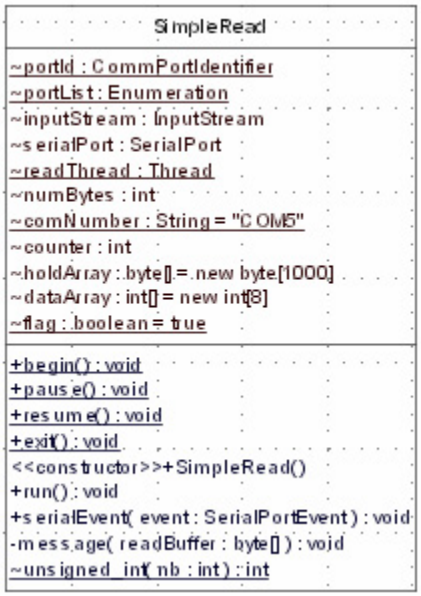



Figure 69. SerialReader class diagram

### 3. Algorithmic Component

The algorithmic component is the main component of the object-tracking application. This component follows the logic mentioned in the above section. The first part of the algorithm consists of the motionDetector class, which is responsible for producing the detection target objects, based on the raw data that the data acquisition component provides. The second part is the set of the three different scenarios classes and implements the actual algorithmic process.

The motionDetector class is the core class of the application and serves multiple purposes (Figure 63 & Figure 70). Its main task is to receive the array that contains the raw WSN data from the SimpleRead class, to filter it, and to create detection outputs. Those outputs contain the raw data that the sensor sent, the object's category based on the sensor's signature, and the default speed, direction, and time to camera values. Those outputs are stored in the "target" objects and then are passed to the proper scenario class, based on the user's choice. The motionDetector class is also responsible for calculating the PIR and magnetic thresholds that are used in the filtering operation mentioned above. In addition, it contains all the data structures (FIFO) that the scenario classes are using. Moreover, it contains utility methods in order to support the operation of scenario classes: it implements the FIFO and calculates the speed, speed history, average speed, and the

arrival time of the object to a specific point near to the TRSS camera's position. Finally, the motionDetector class is responsible for triggering the TRSS system and for informing the user interface component about the application's outputs.

The waitTime class (Figure 71), part of the algorithmic component, is a thread, "utility" Java file that is used to produce the proper time delay whenever the object-tracking application feeds with data and triggers the TSSRv3 system. It is called by the motionDetector class

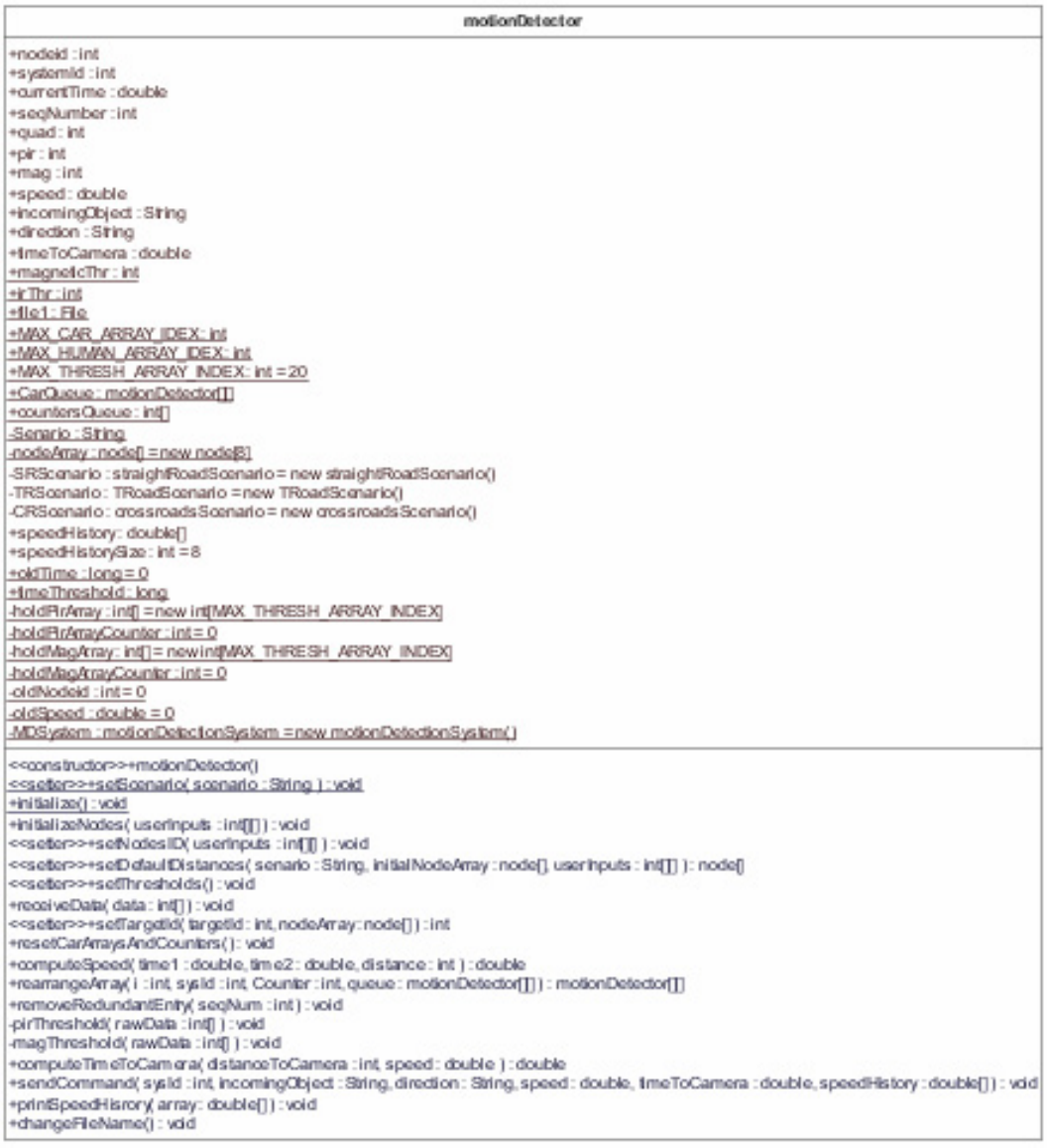



Figure 70. Class diagram for the motionDector class

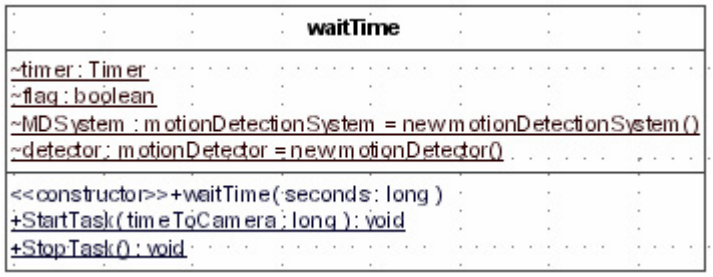



Figure 71. Class diagram for the waitTime class

The second part of the algorithm component is the set of scenario classes. The classes implement the algorithmic manipulation of the detections (target objects), mentioned above in the related section, and produce, with the cooperation of the motionDetector class, the desired outputs. Specifically, they are responsible for producing the object's direction output and for placing the target object in the correct node's data structure. All three scenario classes are similar, containing a long series of selection statements. The basic class is the straightRoadScenario, the rest are developed based on it. Their differences come from the different topologies they serve. Figure 72 presents the scenario classes diagrams.

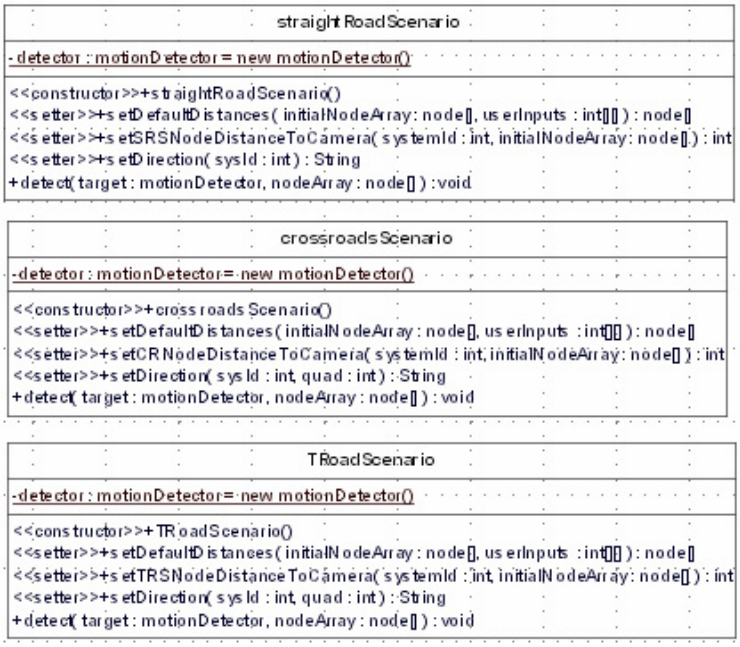



Figure 72. Class diagram for the scenarios classes

### 4. Information Flow

Whenever the configuration of the object-tracking application ends, the flow of the information through the application is as follows: the MSP 410 wireless sensor network senses and returns AM to the system's gateway MBR410CA. The gateway forwards the messages through a serial port to the base station (PC). The SimpleReader class receives the AM and extracts the useful information. Then it forwards the raw data to the motionDetector class. The motionDetector class filters the data identified and produces detection objects and identifies what kind the detected object is. Then it forwards the target objects to the selected scenario class. The scenario class implements the algorithmic process and, with the motionDetector and waitTime classes' cooperation, produces the required outputs (e.g., direction, speed). Those outputs are sent to the user interface where they displayed. In addition, if the application is cooperated with the TSSR system, the motionDetector class triggers the TSSR, which takes the pictures and then forwards them to the FTP server. Figure 73 demonstrates the information flow by using the application's component description, while the next section summarizes the object-tracking application's products.

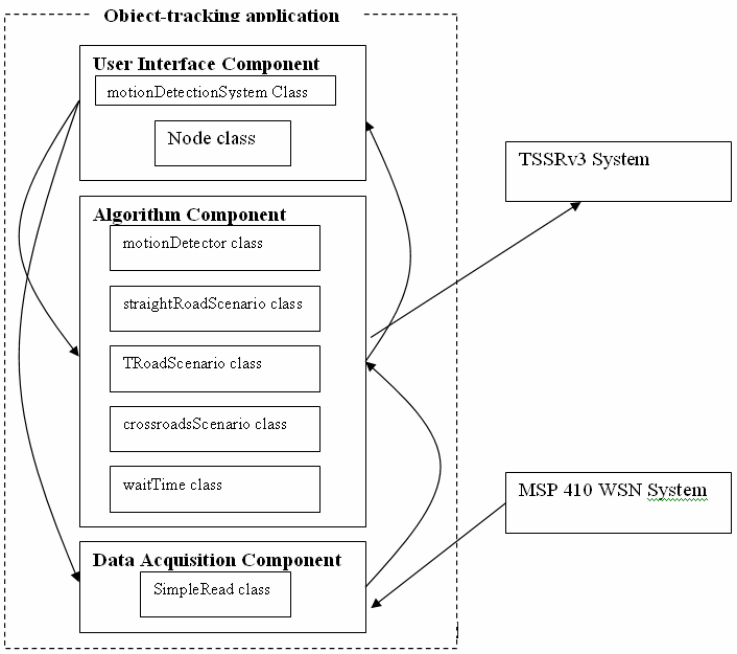



Figure 73. Information flow inside and outside the object-tracking application

## 5. Object-Tracking Outputs

As mentioned above, the outputs that the object-tracking application produces are: the direction, based on the specific scenario that the application currently runs; the speed, based on the time difference between sequential detections; and the speed history and average speed, produced by holding historical detection data. Finally, the timeToCamera output specifies, based on the object's current direction and speed, the object's arrival time at a specific point. All the above outputs are produced or updated per detection and displayed in the right window of the application's GUI (Figure 74). In addition, a command line window (Figure 76) is used to display all the outputs and the raw values that the object-tracking application receives from the MSP 410 wireless sensor network. Finally, Table 5 summarizes all the outputs per scenario.

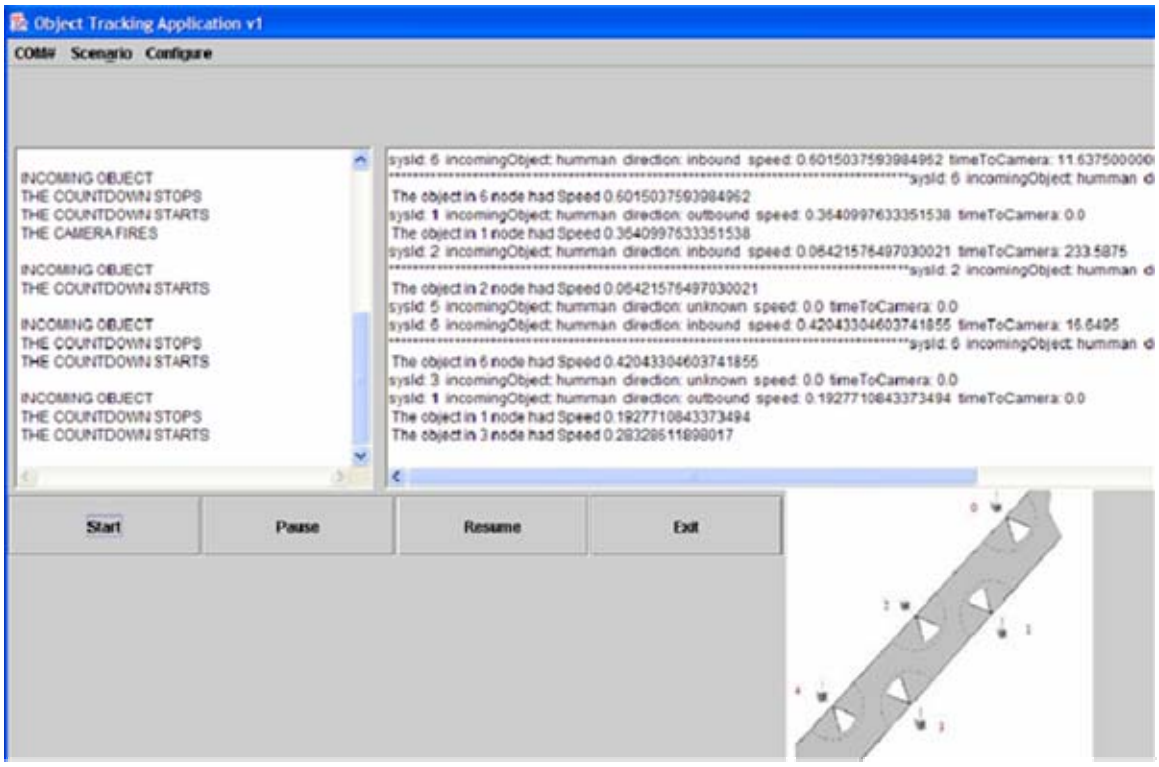



Figure 74. Object-tracking application's output in the GUI window

| Object-tracking Application's Outputs | | | | | | |
|---|---|---|---|---|---|---|
| **Scenario** | **Object's id** | **Direction** | **Speed** | **Speed History & average speed** | **Estimate Arrival Time** | **Commands** |
| **Straight-road** | Human<br>Vehicle | Inbound<br>Outbound | m/sec | Nodeid for all the nodes that detected the object and the corresponding speed.<br>Computes the average speed<br><br>nodeid# m/sec<br>m/sec | sec | Selected scenario<br>Start<br>Pause<br>Resume<br>Exit<br>Camera fires |
| **T-road** | Human<br>Vehicle | Inbound<br>Outbound<br>Left to Right<br>Right to Left<br>Inbound Left to Right<br>Outbound Left to Right<br>Inbound Right to Left<br>Outbound Right to Left | m/sec | Nodeid for all the nodes that detected the object and the corresponding speed.<br>Computes the average speed<br><br>nodeid# m/sec<br>m/sec | sec | Selected scenario<br>Start<br>Pause<br>Resume<br>Exit<br>Camera fires |
| **Crossroads** | Human<br>Vehicle | Inbound<br>Outbound<br>Left to Right<br>Right to Left<br>North to South<br>South to North<br>Inbound Left to Right<br>Outbound Left to Right<br>Inbound Right to Left<br>Outbound Right to Left<br>Inbound North to South<br>Outbound South to North<br>Left to Right North<br>Right to Left North<br>North to South Left<br>North to South Right | m/sec | Nodeid for all the nodes that detected the object and the corresponding speed.<br>Computes the average speed<br><br>nodeid# m/sec<br>m/sec | sec | Selected scenario<br>Start<br>Pause<br>Resume<br>Exit<br>Camera fires |

Table 5. Object-tracking application's outputs

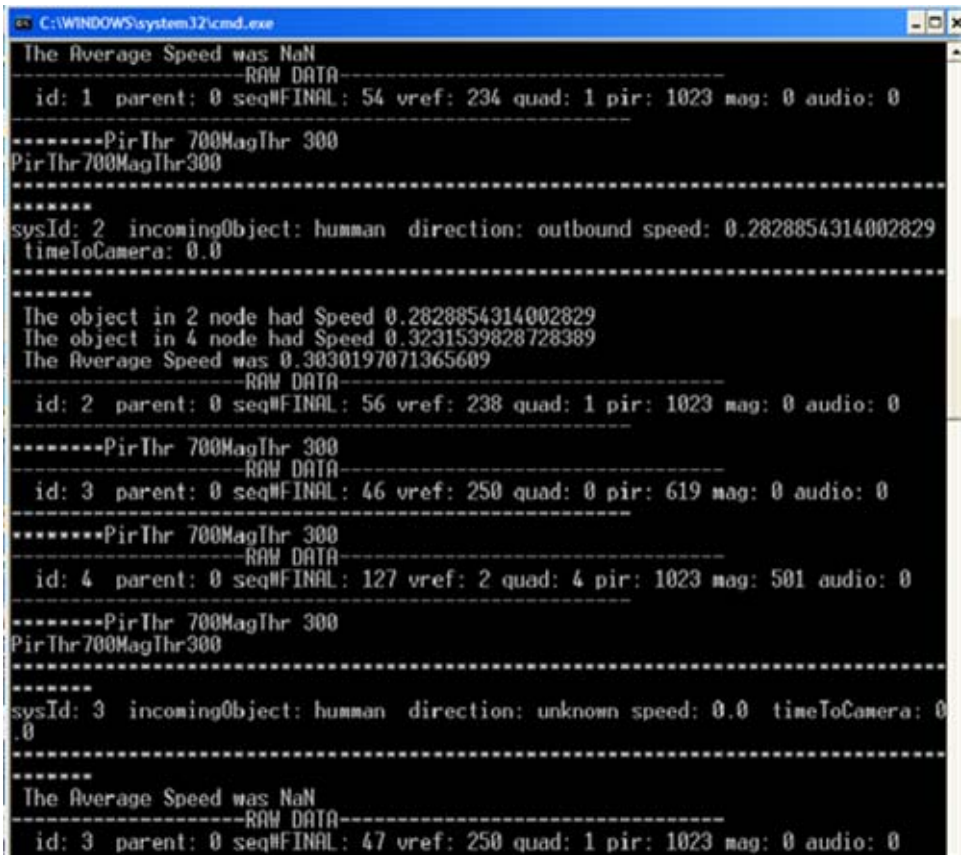



Figure 75. Object-tracking application's output in the command line window

The purpose of this chapter was to present the main objective of the object-tracking application. It describes the motivation, the application requirements, the design and programming issues, and the application's software components. The object-tracking application is a complete and useful application, sited in the systems control station, whose purpose is to demonstrate a real-world implementation of the wireless sensor network systems. The next chapter continues with the system testing and evaluation.

# V. TESTING AND EVALUATION

## A. HARDWARE TESTING AND EVALUATION

One of the important initial steps for the object-tracking application implementation was to determine the way that the application should use the MSP 410 wireless sensor network system. Because the information that Crossbow provides in its manuals is insufficient, we had to performe additional, primitive hardware testing. The purpose of the testing was to identify some of the required wireless sensor network system characteristics in order to use them in the object-tracking application implementation and, later on, as the application deployment guide. Thus, we focused our tests in an attempt to identify two different characteristics: the maximum RF distances between the nodes and the sensors characteristics. The tests were performed outdoors in an uncontrolled environment but are adequate for our purpose. The following sections present the test results.

### 1. RF Range Test

The maximum RF distance tests were performed outdoors across a field. We placed the sensors at a 44 cm height from the ground. The testing height was based on the object-tracking application requirements to detect humans and cars. During the tests we used the xserve and the MOTE-VIEW Crossbow applications to evaluate the connectivity between the nodes that we used. We performed all the tests under similar conditions: on sunny days with temperatures around 20 degrees Celsius and low humidity. In addition, we used new batteries for the system's nodes.

We followed two different approaches for the tests. For the first one we initially placed the nodes close to each other and to the MSP 410 gateway and we left them be connected. Then we moved both the sensors 15 meters away from the gateway and we checked the connectivity. Both of them, as we had expected, were connected directly to the base station. The next steps were to increase the distance between the two nodes. Initially, we used five meter steps, later two meter steps, and finally one meter until we lost the connectivity. We performed the test five times; the average maximum distance between nodes that we achieved was 65 meters.

In the second set of tests, we used a quite different approach. We first operated only the gateway and one of the nodes: when the node connected to the gateway, we again moved the node a distance of 15 meters away from the gateway. Then we placed a second node 65 meters away from the first node (the maximum average distance achieved in the previous experiment), and we operated it. We performed this experiment also five times. In all of them, the nodes did not establish connection in a reasonable amount of time (15 minutes). Then, we started reducing the distance between the nodes, using two meter steps and a ten minute waiting time. The average distance at which the nodes finally connected to each other was 45 meters.

After evaluating the experimental results of the above two tests, based on the object-tracking requirements, we concluded the following. It seems that it is easier for the MSP 410 nodes to maintain their connectivity than to establish a new one when the distance between them is more than 45 meters. The object-tracking application requires each node to maintain connectivity with at least two neighboring nodes on each side. Thus, we conclude that a reasonable deployment distance between the nodes, placed at a height of 40cm from the ground, that serves the application requirements, is 20 meters. This deployment allows the system to maintain the connectivity even if it loses an intermediate node. In addition, if a node must be replaced by a new one, the new node will be connected in a reasonable amount of time.

**2. Environmental Influence on the PIR Returns**

A preliminary experiment that we performed was related to the return PIR values from the environment. The purpose of this experiment was to identify how the environment temperature affects the return PIR values. We performed this experiment using four different MSP410 nodes, which we placed in a solitary outdoor position to avoid returns from objects detections. Two of the nodes were from the old MSP 410 system (Crossbow claims that it was an experimental system); and the remainder were from the new set that we used in our deployment. We operated the nodes for more than 24 hours and collected the data through MOTE-VIEW. Figure 76 presents the return PIR values from one of the new nodes. Figure 77 completes the above Figure 76 by displaying the temperature changes during the test period. All the return patterns were similar, although the new MSP 410 set nodes seemed more stable. The analysis of the

PIR patterns indicated that the environmental temperature has a considerable affect on the PIR values.

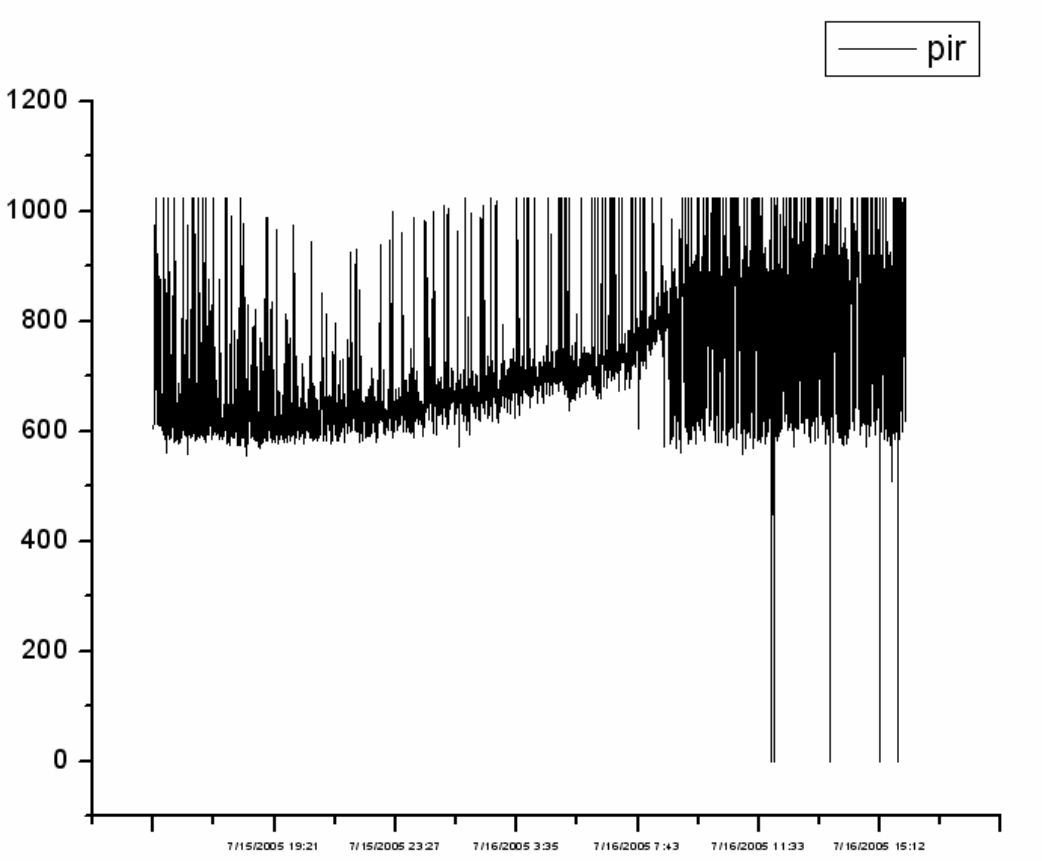



Figure 76. Fluctuation of the returned PIR value

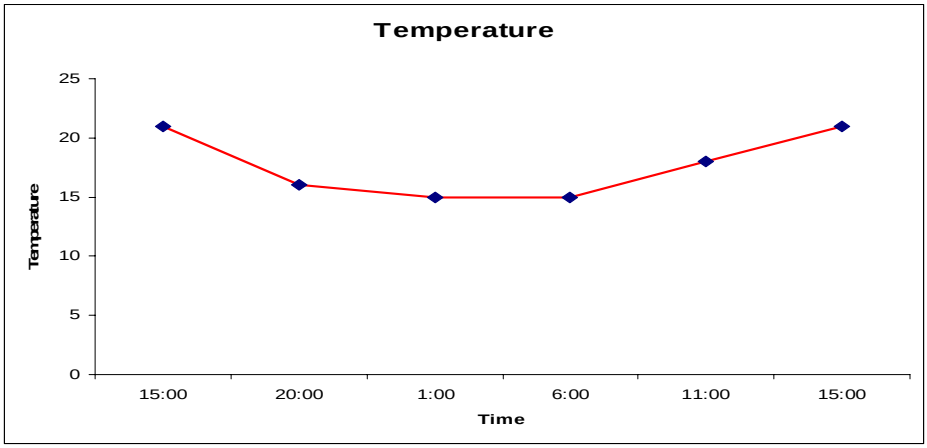



Figure 77. Temperature change

After evaluating the results of the above experiment, we concluded that, especially for outdoor deployments of the MSP 410 wireless sensor network system, the environmental temperature affects the return PIR values. During the development of the object-tracking application, we used this information to design our PIR threshold mechanism. The experimental results agree with a Crossbow verbal confirmation that the PIR sensors of the MSP 410 system are affected by the environmental temperature. Crossbow also confirmed that the influence is greater if wind is blowing in the area of deployment.

We performed similar experiments for the magnetic sensor, to identify how it responds when the magnetic field is changed. It seems that the magnetic sensor is more capable of calibrating itself to fit the environmental conditions. Thus, if the changes in the magnetic field are slow, the sensor is able to adjust its internal threshold level and to avoid false returns. The object-tracking application also uses its own magnetic threshold to improve the application's response during the period that the sensor recalculates its internal threshold.

### 3. Sensor Sensing Range and Detection Probability

To add to our knowledge about the optimum sensing ranges of the MSP 410 nodes, we performed two experiments. Crossbow also, in its manuals, refers to the sensors' sensing distances (chapter 3), but the information that it provides is not sufficient for a real-world deployment. We designed two experiments to detect different types of objects. In the first experiment, the node detected vehicles, in the second, humans. We used the same node for both experiments, to avoid any differences between nodes. We used outdoor deployment in both situations, and we conducted the tests under similar environmental conditions: on sunny days with temperature of 20 degrees Celsius and calm winds . The node's position was at a height of 44 cm and only one of the node's PIR sensor monitored the object path. As a monitoring tool, we used the object-tracking application for evaluation of the results.

#### *a. Vehicle Detection Experiment*

In the first experiment, we used vehicles as our target objects for identification. We placed the node close to the road and gradually increased the distance from it in one meter steps. For the target, we used the normal road traffic. During the first

part of the experiment we tried to identify differences in the sensors' returns for different vehicle types: we did not notice a great fluctuation. Thus we decided to continue using the road's normal traffic as our target. The cars' speed was approximately 32 km/h (20mph). We detailed PIR, quad, and magnetic returns from the node. In addition, we kept records of the node's distance from the targets and the number and type of detections that the node returned. Each step of the experiment was conducted 15 times. Figure 78 and Figure 79 present the changes in the return values as the distance from the target was increased. Figure 80 demonstrates the variation in the number of detections as the distance increased.

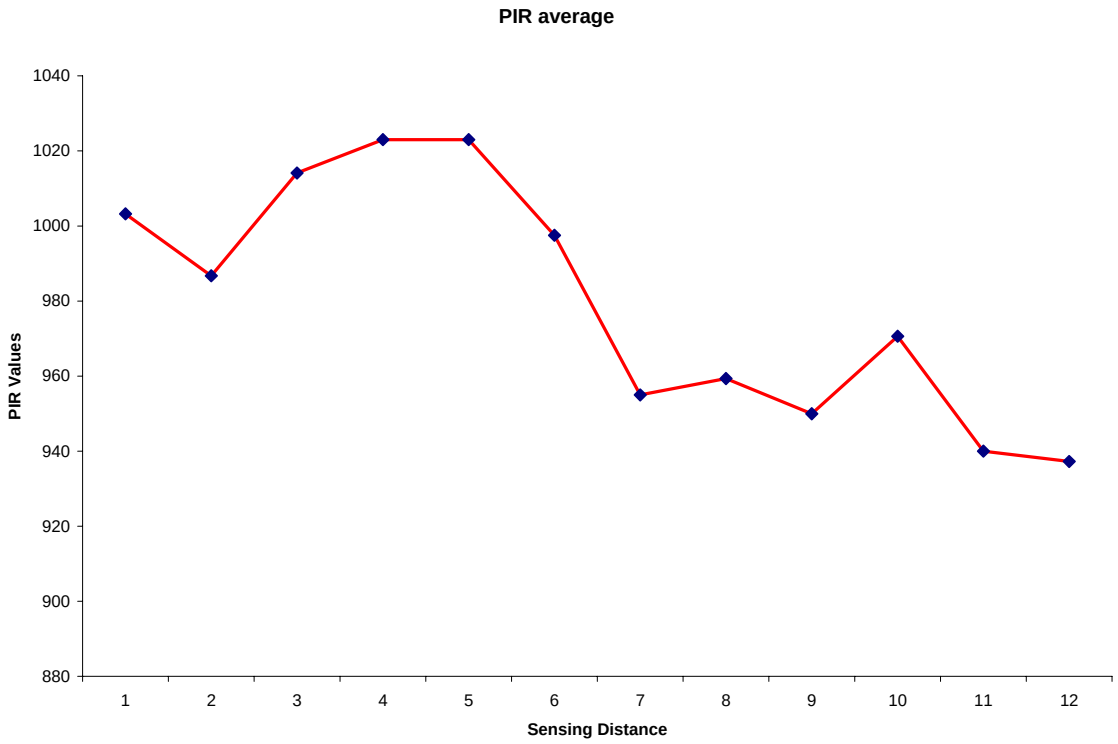



Figure 78. Average returned PIR values per distance from a car's path

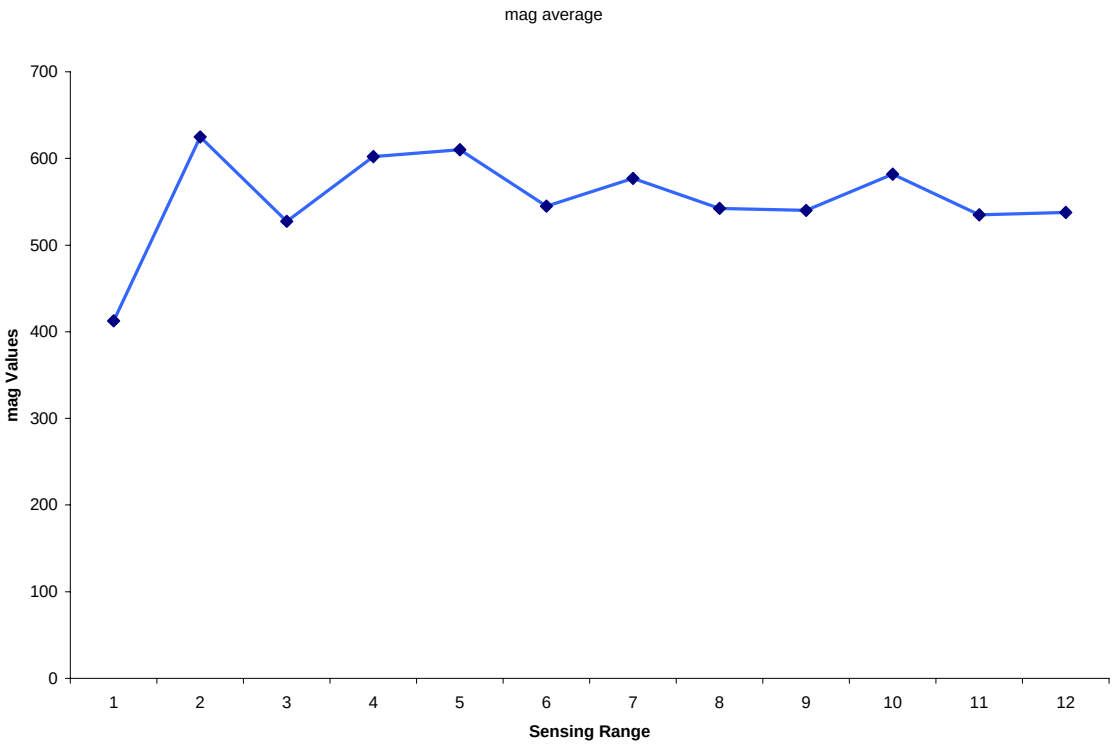



Figure 79. Average returned mag values per distance from a car's path

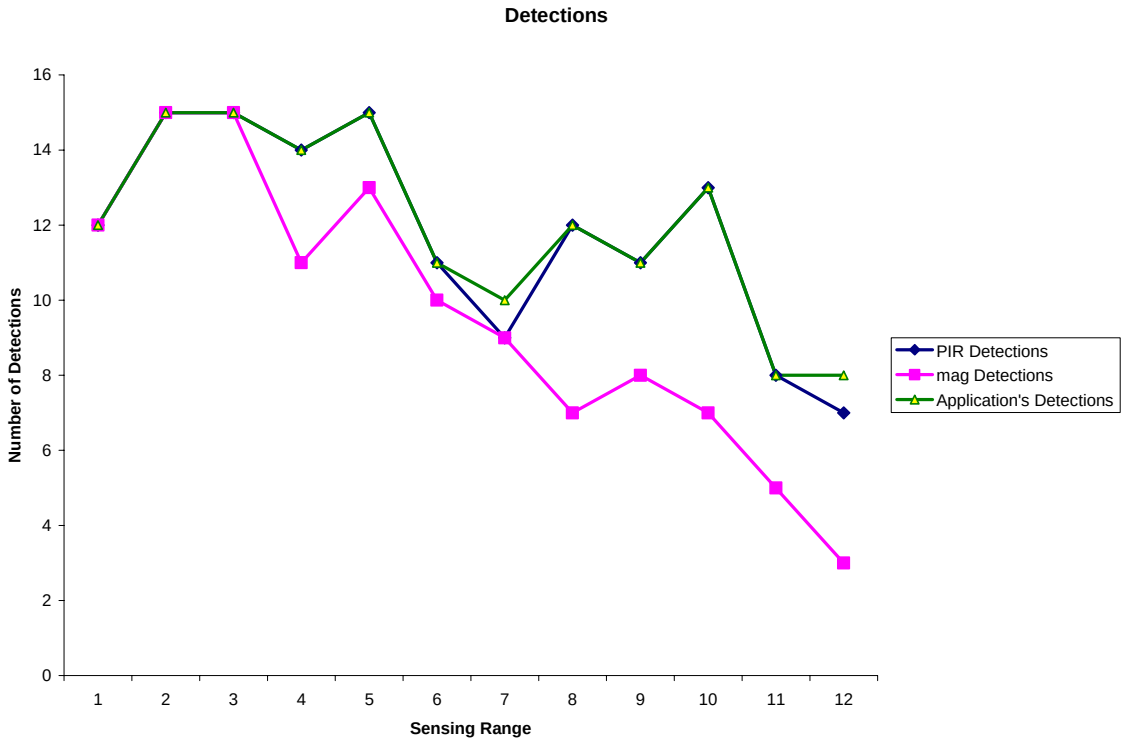



Figure 80. Change in the number of detections as the distance from a car's path increased.

After evaluating the above experimental results, we concluded the following: In general, as the distance from the object's path increased, the returned PIR and magnetic values decreased. In addition, the quad value that is tied up with the PIR detections became unstable and the detection probability decreased. The returned PIR and magnetic values are the highest for a range of distances between two to six meters from the object's path. For those distances also, the quad value was stable and the detection probability had its highest values.

From very short distances, less than two meters, the nodes were not very sensitive. Although this result seems strange, it happens because of the way the sensors work, especially the PIR. The PIR sensor uses a set of sensing beams and monitors the changes between them to determine detection. From very short distances from the object, those beams and the sensing field of view are too narrow and thus not very accurate. This also explains why the sensor for large distances is not efficient enough to detect more than one car going the same direction at the speed, following one another at a reasonable distance. In that situation the projection of the sensing beam onto the object's path (road) is very wide, so it is difficult for it to return two detections for two cars close together. Figure 81 highlights this issue.

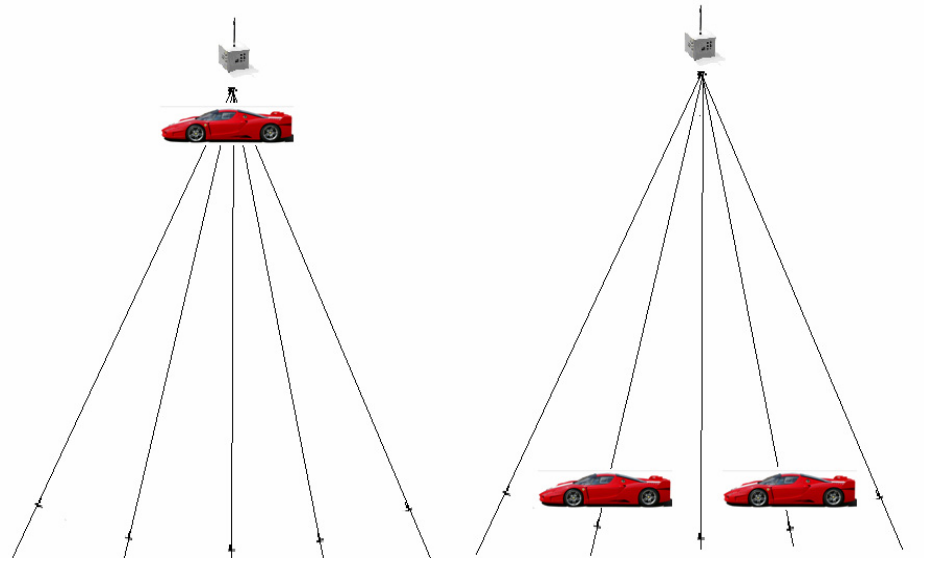


Figure 81. Node topologies that affect the system's performance.

Our conclusion from this experiment was that the returned magnetic values are more stable, compared to the PIR values, as the distance increases; but the PIR sensor manages to return car detections from greater distances. In addition, the object-tracking application evaluates the returned messages by using all the PIR, quad, and mag values to determine a detection: this seems most effective when it returns the maximum number of detections, as compared to the number of detections per sensor.

### *b. Human Detection Experiment*

The second experiment was to detect humans. We used the same set-up and plan as in the car detection experiment. In this experiment, each run was to detect the same target human. The target's speed was around 5 km/h (3 mph) and the human's height was 1.75 meters, and the environmental conditions were similar to those of the car detection experiment. During this experiment we monitored only the PIR and quad values, because the human did not carry any metallic objects. The following figures present the experimental results from this experiment. Figure 82 illustrates the returned PIR values per distance, Figure 83 the total amount of detections per distance.

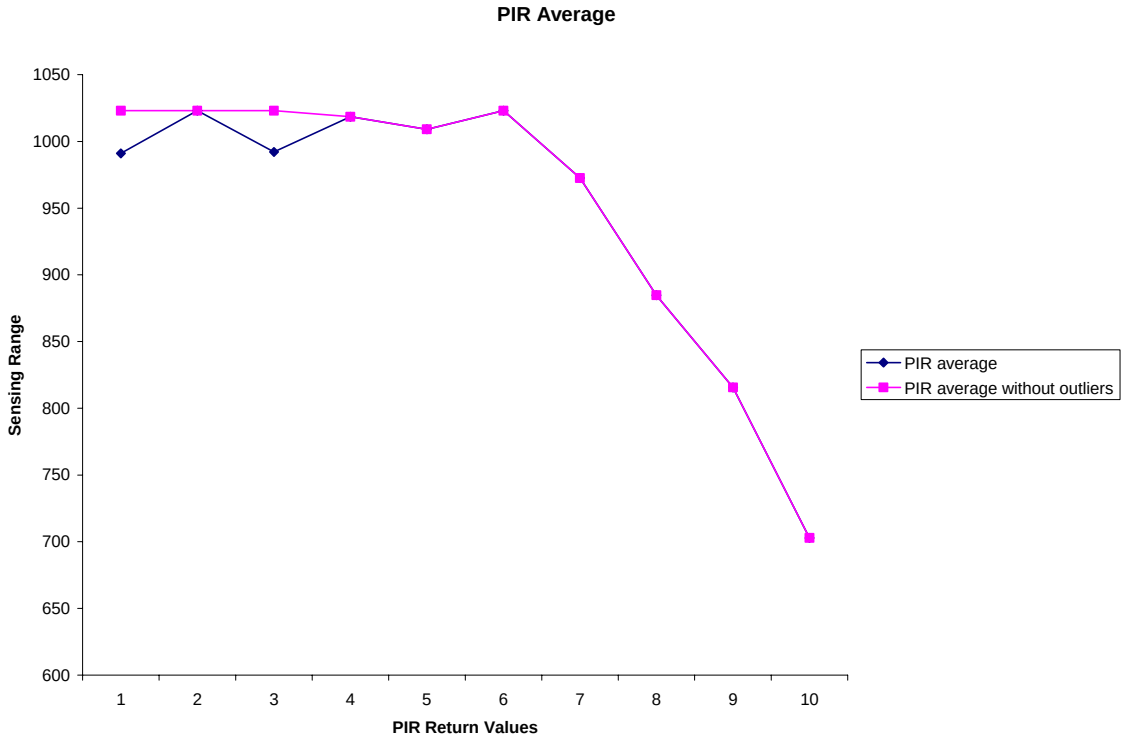



Figure 82. Average returned PIR values per distance from the human's path

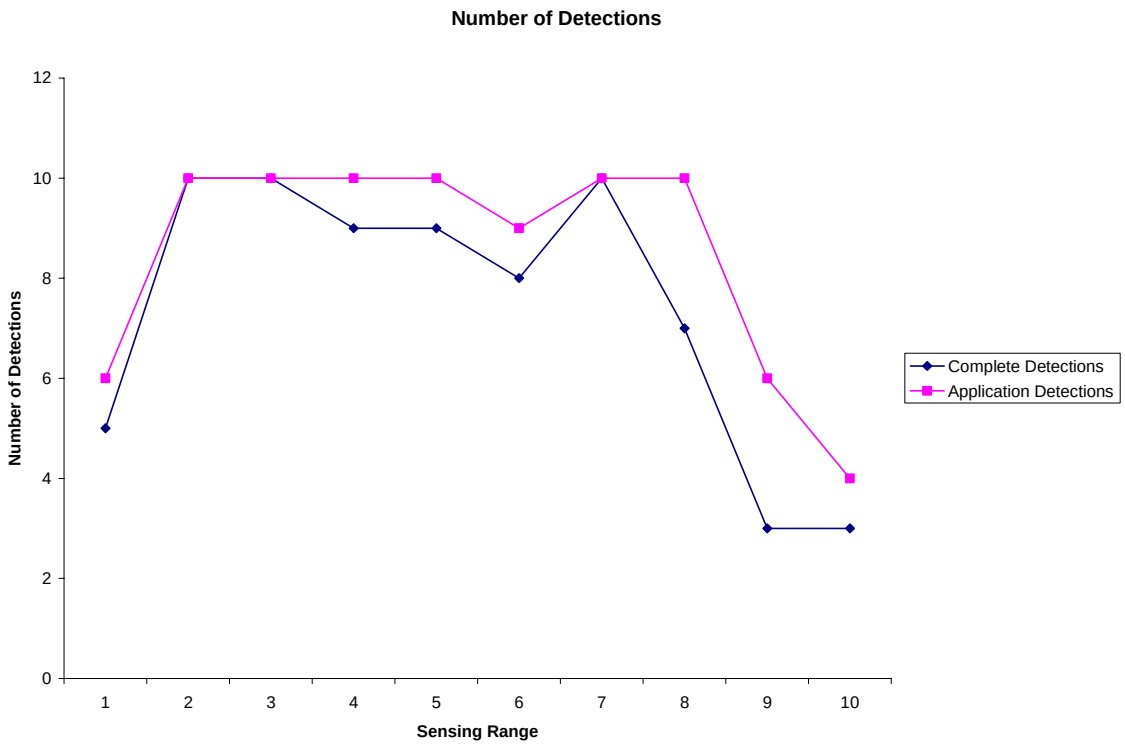



Figure 83. Change in the number of detections as the distance from the human's path increased

In evaluating the results of this experiment, we identified similarities with the car detection experiment. For short distances, here also the PIR sensor is not very sensitive, but for a range of distances from two to six meters the PIR and the quad values are stable. In this experiment, the effective range of the PIR sensor is shorter than in the car detection experiment. One interesting observation is that, for large distances in the car detection experiment, the PIR sensor was not able to separate objects that were close together, while in the human detection experiment for the same situation the PIR sensor returned more than one detection.

Finally, during all the experiments that we performed using the MSP 410 wireless sensor network system, we observed that the node's battery level significantly affected the system's characteristics. Although we did not perform a specific experiment about the battery level, it seems that it affects not only the node's sensing characteristics, but also its communication characteristics.

All the above experiments are important for the object-tracking application development. They improve the description of the MSP 410 system's characteristics that the Crossbow manuals include. In addition, these experiments helped us design many of the application deployment plans that relate to the node sensing characteristics, environmental effects, and topological considerations. To complete the testing and evaluation discussion, the next section presents the results of additional experiments with the object tracking application.

## B. OBJECT-TRACKING APPLICATION: TEST AND EVALUATION

The preceding section describes the tests that we performed on the MSP 410 wireless sensor network system. The purpose of this section is to discuss and evaluate the object-tracking application based on additional tests that we performed. This section also provides recommendations for the application.

### 1. Evaluation of the Object-Tracking Application

The evaluation of the object-tracking application is based on the tests that we performed, especially the final steps, where we evaluate the whole system. We also performed two "official" tests of the application within the context of the TNT field experiments. We performed the application evaluation by comparing the experimental results and theoretical capabilities of our product with the initial requirements and application assumptions discussed in previous chapters.

The application was developed to demonstrate a real-world use of a wireless sensor network in object tracking. In evaluating the final product with respect to its intended purpose, we conclude that the object-tracking application was successful. Although it is a research prototype and not a commercial product, our application is a simple, stable, stand-alone system capable of demonstrating an important application of WSNs. It takes raw data from the WSN, aggregates it, evaluates it, and based on the evaluation, produces detection alerts and tracks the object to provide motion outputs.

Although it can be easily used on top of other WSN products, the object-tracking application was developed for the MSP 410 WSN and uses its characteristics to produce its outputs. The application reliability is mostly based on the MSP 410 reliability. Although we used techniques to increase the application redundancy and reliability, the application is dependent on the node returns. If the WSN does not return proper signals,

the application will have difficulty in producing reliable outputs. During the application tests, we identified differences between the old and the new MSP 410 systems that we had available. The new system seems more stable, probably because it uses better the internal thresholds, especially for the PIR sensor. We saw great differences in the application performance during the transition from the old to the new Crossbow system.

The preceding sections describe the experimental results of the MSP 410 system. The object-tracking application used them to increase its reliability. Its use of the proper scenario selection, optimum topology of the eight available WSN nodes, and the efficient processing of the returned data increased the system's overall performance. Figure 81, and Figure 84 present the application enhancement of the detection probability per node. The additional enhancement is provided by the object-tracking application because of the redundancy that it supports. As is described in chapter 3, the application reacts efficiently and produces data even if the WSN loses a number of nodes. In addition, it is able to handle situations in which one or more nodes do not detect the object. In general, the application redundancy and reliability enhance the characteristics of the WSN and make the entire system more attractive.

The decision for data manipulation at the base station was another aspect of the application that had to be evaluated. Based on the MSP 410 nodes' characteristics and also the limited number of nodes that the application had available, the above choice seems reasonably.

The final aspect of the application that we evaluate is the use of WSN for object tracking. As chapter 2 shows, the WSN is an efficient solution for an application that requires long-lasting deployment and serves low-rate data transmission from the environment. Given this, we assumed that the object-tracking application can be deployed indoors in a corridor or outdoors  by a remote road without heavy traffic. Thus, the application assumes that there is one object being tracked at a time by the system. The final product oversubscribes that assumption providing accurate outputs for more than one object inside the system if they are going in the same direction Their order inside the system does not change and the distance between them allows their resolution by the MSP 410 (the current application's configuration support up to three objects at the same

time inside the system. The system's capacity can be easily increased). In addition, under some circumstances the application is able to track more than one object inside the system having different directions (especially in the T-road and crossroads scenarios).

The application's algorithm that chapter four describes promises that every moving object inside the system will be detected and tracked. The algorithm actually requires two or three nodes in the row to produce the outputs. The way the algorithm works and the fact that there are 8 available nodes assures the success of the application. The already high detection probability that the MSP 410 provides is enhanced by the application. The experimental results of the final product indicate that the application provides a high probability for object detection and tracking at speeds up to 72km/h (45 mph). Figure 84 presents pictures taken during the final tests of the application, in which the car's speed is 64 km/h (40mph). The scenario was straight-road, we used six MSP 410 nodes, and their topology was along both sides of the road. The road was five meters wide and the distance between the nodes was 20 meters. The nodes were 44 cm high, on top of the card boxes displayed in the pictures. The weather was cloudy, without wind, with a temperature of 18 degrees Celsius. During this test we tested the application at three different speeds: 16km/h (10mph), 32 km/h (20mph), and 64km/h (40mph). For each speed we made five runs. The application detected and tracked the car in all runs.

Figure 84. Pictures taken during the final object-tracking application tests: Fort Ord California, August 2005.

The TNT exercises that NPS performs every quarter at Camp Roberts, California, gave us two additional opportunities to test and demonstrate the object-tracking

application. The first opportunity was the TNT experiments on May 24-25,' 2005, in Camp Roberts and NPS, respectively. In both tests, the deployment was outdoors along two sides of a road on sunny days, the distance between the nodes was 20 meters, and the nodes' height was 25 cm. In both experiments we used the old MSP 410 system and in both experiments the object-tracking application cooperated with the TSSRv3 system. The temperature for the first test was around 25 degrees Celsius, and for the second, around 20 degrees Celsius. The test at Camp Roberts was a controlled experiment. The traffic was low and controlled, in order to evaluate the system's performance. The deployment environment was similar to the environment in which the application is expected to be used. Figure 85 presents photos that the TSSRv3 took during the experiment and transmitted to the NPS ftp server through satellite communications (Globalstar).

Figure 85. Pictures taken by the TSSRv3 system during the application's test at Camp Roberts, California, May 2005

The next, day May 25, we performed the same experiment with the same application configuration on the NPS campus. During that test the deployment was along an NPS road where the traffic was not controlled. Figure 86 presents pictures that the TSSRv3 camera took during the second experiment and transmitted to the NPS ftp server through a wireless 802.11b connection. Although the application development was in its initial phase only the data capture and the detection-evaluation parts of the application

were available at that time, the results were encouraging. The application was able to trigger the TSSRv3, even though the WSN was the old and quite unstable MSP 410 system.

Figure 86. Pictures taken from the TSSRv3 system during the application's test at NPS, Monterey, California, May 2005.

The third opportunity to test the object-tracking application in the context of a TNT exercise was on August 30, 2005 at Camp Roberts. This test was an indoor deployment. We placed the six available nodes in a way that simulated a corridor inside a building, using four meter distances between the nodes. The height of the nodes was 40 cm, on top of a wooden chair. The application's topology was such that the "inbound" direction was from the back of the building toward the front door. At the front door we placed the TSSRv3 camera that the object-tracking application triggers. Although the purpose of the experiment was not to identify motion inside the building, but only to detect humans inside the building, the results were satisfactory. The application managed to detect and track a human in motion inside the building, as Figure 87 indicates. In addition, with minor changes in the program, the application succeeded also in its primary purpose, to identify and trigger the camera whenever it detected humans inside the building, as Figure 88 demonstrates. In both cases the pictures from the camera were transmitted from the TSSRv3 system via cellular phone to the ftp server sited in the NPS.

The TNT experiment indicates that the application is stable enough also for indoor deployment. In addition, it demonstrates that the object-tracking application, with

minor changes, can be used as a general detection system, a use that is simpler than motion detection and tracking.

Figure 87. Pictures taken by the TSSRv3 system during the application's test at Camp Roberts, California, August, 2005

Figure 88. Pictures taken by the TSSRv3 system during the application's test at Camp Roberts, California, August, 2005

For all the scenarios except the straight-road, we performed mostly indoors tests. Those scenarios used the same algorithm and concept as the straight-road scenario. The direction was the major difference from the straight-road scenario; the speed and the calculation for the other outputs are the same as in the straight-road.

The above evaluations and experiments indicate that the object-tracking application succeeded in its initial requirements as a wireless sensor network demonstration application.

## 2. Object-Tracking Application Deployment Recommendations

The preceding sections provided the application description and the test results. This section will provide deployment recommendations for users of the object-tracking application to optimize the application results. The two important steps that the user must do are the scenario selection and nodes topology, for both indoor and outdoor deployments. The selected scenario must be close to the real deployment environment. The following paragraphs present recommendations for the nodes topology.

Distance between the nodes is the first issue. The distances have to be appropriate so that they fit within the available deployment space. The user must keep a balance between the number of deployed nodes and the distance between them. The minimum number of nodes that the object-tracking application requires is three; the maximum that it can handle is eight. The more nodes that the application uses, the more reliable and accurate the results produced. The application prefers to use large distances between the nodes. The maximum deployment distance that we recommend, which depends also on the nodes' height, is 20 meters for a node-height of around 40cm from the ground in moderate vegetation. This distance provides redundancy in the MSP 410 system, and thus, its ability to respond and handle node failures. In addition, the larger the distances between the nodes that the application uses, the more accurate and reliable the results. Large distances provide the application with enough time to collect and process the data, in order to produce the desirable outputs. Finally, the user must be careful not to place the nodes too close to each other, because sensing-area overlapping is not desirable for the application.

Node height is another deployment parameter that affects the application's performance. The nodes' height affects the communication distance, but more importantly, it affects the sensors' performance. The nodes' height must be adjusted based on the kind of object that the object-tracking application must detect and track. It is preferable for the PIR to be able to sense the main heat source of the object (e.g. a vehicle's engine); similarly, for the magnetic sensor, it is better to monitor the main metallic mass of the object. Careful node-height adjustment maximizes the sensors' returns and ensures the RF communication.

During the system's deployment, one or both sides of the road can be used. The choice mainly depends on the road's width and structure. Our recommendation is to deploy the system in a narrow road where all the nodes have an equal probability to detect and track the moving object. In sum, the choice is up to the user: the important issue is to provide the sensors an equal detection probability.

The distance that the nodes have from the object path is another issue with which the user must take extra care of. As the experimental results indicate, the distance of the nodes from the object affects the application performance. Very large distances and very short distances fail to optimize the node sensing characteristics and the application's overall use. Our recommendation is to use a node-distances range from two to five meters from the expected object's path.

Additionally, we recommend deploying the system on a road or corridor where there is light traffic. According to the application assumptions, it is capable of handling multiple objects going the same direction (the current configuration supports up to three).

Finally, we recommend the user to follow the instructions that the application user interface provides during the application's configuration. In addition, we recommend that, after the end of the configuration phase, leave the system a brief period of time to produce and calibrate the thresholds it will use.

The purpose of this chapter was to continue the object-tracking application overview describing the application's testing and evaluation phase. We evaluate and present the experimental results in an objective way. This chapter initially described the test that we performed, first, to evaluate better the Crossbow MSP 410 wireless sensor network system, and second, to have more robust inputs about the MSP 410 capabilities during the object-tracking application's development. The second part of the chapter evaluated the application by analyzing the efficiency of the application implementation and providing experimental results. In the final section the chapter provides the user with deployment recommendations in addition to those that the application user interface provides. The next chapter is contains our overall conclusions and suggests future improvements that can be made to this project.

THIS PAGE INTENTIONALLY LEFT BLANK

# VI. DISCUSSION

## A. SUMMARY AND CONCLUSIONS

In this thesis we have explored the use of wireless sensor networks for object tracking and motion estimation. We described the revolution in wireless communication and important wireless implementation techniques and protocols. We introduced the wireless sensor networks, theoretical characteristics and system constraints, and current available and possible networking architectures and deployment topologies. We also described some of the related standards. In chapter two we presented an overview of the wireless sensor networks.

We continued in the second part of this thesis with a description of the hardware and software products that we used. We focused on Crossbow wireless sensor network products, by describing their characteristics and the functionality that they support.

In the third part of this thesis, we analyzed the object-tracking application which is build on top of the Crossbow MSP 410 wireless sensor system. The object-tracking application is an event-driven application sited in the base station. This project aimed to demonstrate a real-world application that uses a WSN to communicate. Chapter four presented the object-tracking application requirements and how our work implemented them. It also analyzed the design, the logic, and the algorithm that we developed to process the data returned from the wireless sensor network. It explained analytically how the application produces detection signals and tracks the object's motion. We explained the selected scenarios and demonstrated the suitable node topology for the application. Chapter four also described the active message format that the MSP 410 system uses and how we used the information that it contains. Further, it described the assumptions that the designers make before the programming phase and analyzed the structure and the functionality of the application's software components.

The object-tracking application description continued in chapter five with the system's testing and evaluation. It describes additional tests that we performed on the MPS 410 wireless sensor system's communication and sensing characteristics and

evaluated the experimental results in the context of the object-tracking application. The chapter also explains the application testing and evaluation and concludes with deployment recommendations.

Finally, we conclude our study of the wireless sensor network field with the observation that it is a promising new technology. It could be a way to achieve ubiquitous computing and embedded Internet. It seems an efficient solution for many applications that involve deep monitoring of a deployment environment.

## B. FUTURE WORK

The object-tracking application that we propose is by no means complete in every respect. Future work should include thorough testing of the MSP 410 wireless sensor network to produce more accurate and complete information about WSN characteristics and capabilities. The object-tracking application would also benefit from additional testing to evaluate all the possible deployment scenarios that it may cover and the possible environments within which it can be deployed.

A weakness of the object-tracking application is its requirements. The object-tracking application was built as a demonstration application on top of a specific wireless sensor network system with a limited number of nodes. As an initial study, the object-tracking application uses the simplest programming solutions. Although simple and easy to follow, the solutions set limitations on the application efficiency and the maximum number of nodes that the application is able to handle. Thus, future work may update the application software components to achieve greater efficiency.

Possible future work could involve different wireless sensor network architectures to efficiently handle a large number of nodes. The current flat network architecture that the MSP 410 system has is suitable only for a small number of nodes. A study about the xmesh network stack that the system uses would demonstrate the actual networking capabilities of the system. Additional future work, based on complete WSN system characteristics and xmesh evaluation results, could involve node programming capable of performing specific tasks and targeting to improve the system's overall performance.

Finally, future work must be done in the application's user interface. An efficient, smart, and user-friendly GUI is important. This GUI will help the user during the

configuration phase by making proper evaluations of the input values, by preventing the user from inserting invalid data, and by avoiding application misbehavior. In addition, it will provide wireless sensor network monitoring capabilities.

THIS PAGE INTENTIONALLY LEFT BLANK

# APPENDIX. OBJECT-TRACKING SOURCE CODE

```
/**
 * <p>Title:       motionDetectionSystem </p>
 * <p>Description: This class is responsible to provide a simple user interface.
 *                 Additionally it provides instruction, and displays the
 *                 object-tracking application's outputs
 *
 * Assumptions:    The objects inside the system do not change order.
 *                 Inside the application exist up to 3 objects per time with the
 *                 same direction.
 *                 Inside the application exist objects with te same direction.
 *             </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.io.*;
import java.util.StringTokenizer;

public class motionDetectionSystem extends Frame {

  JFrame f = new JFrame("Object Tracking Application v1");
  JButton startButton = new JButton("Start");
  JButton pauseButton = new JButton("Pause");
  JButton resumeButton = new JButton("Resume");
```

```
JButton exitButton = new JButton("Exit");
JPanel mainPanel = new JPanel();
JPanel buttonPanel = new JPanel();
public static TextArea commandTextArea = new TextArea();
public static TextArea dataTextArea = new TextArea();

private motionDetector motDetector = new motionDetector();

JLabel iconLabel = new JLabel();
ImageIcon icon;
// data structure to store the nodes' characteristics
public int[][] userInputs = new int[8][3];

public motionDetectionSystem() {
  try {
    jbInit();
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}

private void GUI() {
  //Frame
  f.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  });

  //Buttons
  buttonPanel.setLayout(new GridLayout(1, 0));
  buttonPanel.add(startButton);
```

```
buttonPanel.add(pauseButton);
buttonPanel.add(resumeButton);
buttonPanel.add(exitButton);

startButton.addActionListener(new startButtonListener());
pauseButton.addActionListener(new pauseButtonListener());
resumeButton.addActionListener(new resumeButtonListener());
exitButton.addActionListener(new exitButtonListener());

//frame layout
mainPanel.setLayout(null);
mainPanel.add(commandTextArea);
mainPanel.add(dataTextArea);
mainPanel.add(buttonPanel);

//Image display label
iconLabel.setIcon(null);
mainPanel.add(iconLabel);
//Text areas
commandTextArea.setBounds(5, 65, 300, 280);
dataTextArea.setBounds(315, 65, 900, 280);
buttonPanel.setBounds(0, 350, 650, 50);

iconLabel.setBounds(650, 290, 380, 380);

JMenuBar menuBar;
JMenu menu3, menu1, menu2;
JMenuItem straight, T_Road, crossroads, Default, Custom,
   insert_COM_Number;

//Create the menu bar.
menuBar = new JMenuBar();
//Build the first menu.
```

```
menu3 = new JMenu("COM#");
menu3.setMnemonic(KeyEvent.VK_A);
menuBar.add(menu3);

insert_COM_Number = new JMenuItem("insert_COM_Number");
menu3.add(insert_COM_Number);

menu1 = new JMenu("Scenario");
menu1.setMnemonic(KeyEvent.VK_A);
menuBar.add(menu1);

straight = new JMenuItem("Straight Road");
menu1.add(straight);

T_Road = new JMenuItem("T Road");
menu1.add(T_Road);

crossroads = new JMenuItem("Crossroads Road");
menu1.add(crossroads);

//Build the second menu.
menu2 = new JMenu("Configure");
menu2.setMnemonic(KeyEvent.VK_A);
menuBar.add(menu2);

Default = new JMenuItem("Default");
menu2.add(Default);

Custom = new JMenuItem("Custom");
menu2.add(Custom);

f.setJMenuBar(menuBar);
```

```
insert_COM_Number.addActionListener(new insert_COM_NumberListener());
straight.addActionListener(new straghtScenarioListener());
T_Road.addActionListener(new TScenarioListener());
crossroads.addActionListener(new crossroadsScenarioListener());
Default.addActionListener(new defaultListener());
Custom.addActionListener(new customListener());

f.getContentPane().add(mainPanel, BorderLayout.CENTER);
f.setSize(new Dimension(1250, 730));
f.setVisible(true);

// Provides the initial instructions
dataTextArea.appendText("WELCOME" + "\n" + "\n" +
                "The object-tracking  application is a research project" + "\n" +
                " that aims to demonstrate a real-world use of the " +
                " wireless sensor networks. " + "\n" +
                " It is producing detection and tracking outputs for object" +
                " that are moving inside the application" + "\n" + "\n" +
                " The user has two initial configuration options:" + "\n" +
                " The first is to custom configure the application by  inserting the" +
                " system's configuration locally following the instructions." + "\n" +
                " The second choice is to use the configuration values that are" +
                " included in the configuration file." + "\n" + "\n" +
                " For the first choice the user uses the menu options and starts inserting
the" +
                " serial port COM# ('COM#' menu otpion)," + "\n" +
                " then it continues by selecting the deployment scenario under the
'Scenario' menu item," + "\n" +
                " finally, it follows the instructions that appeared in the GUIs window "
+ "\n" +
                " and selects the 'Custom' choice under the 'Configure' menu item." +
"\n" +
                " For the second option the user just chooses the 'Default' option" +
                " under the 'Configure' menu." );
```

```
}

public static void main(String[] args) {

  motionDetectionSystem theMotionDetectionSystem = new motionDetectionSystem();

  theMotionDetectionSystem.start();
}

public void start() {
  // build the GUI
  GUI();
  // Initialize the nodes' characteristics
  setDefaultNodeData();
}

public void printCommands(String data) {
  commandTextArea.appendText(data);
}

public void printData(String data) {
  dataTextArea.appendText(data);
}

class startButtonListener  implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // initializes the counters
    motDetector.initialize();
    // Opens the serial port and starts reading
    SimpleRead.begin();
    // Displays the choice
    commandTextArea.appendText(" The Object Tracking application starts!" + "\n");
    System.out.println("Start Button pressed !" + "\n");
```

```
  } // end action performed function
} // end inner class startButtonListener

class pauseButtonListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Pauses the serial port reader
    SimpleRead.pause();
    // Displays the choice
    commandTextArea.appendText("pause Button pressed !" + "\n");
    System.out.println("pause Button pressed !" + "\n");
  } // end action performed function
} // end inner class pauseButtonListener

class exitButtonListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Terminates the application
    SimpleRead.exit();
    // Displays the choice
    commandTextArea.appendText("The Object Tracking ends!"+ "\n");
    System.out.println("exit Button pressed !" + "\n");
  } // end action performed function
} // end inner class exitButtonListener


class resumeButtonListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // restarts the application it is paused
    SimpleRead.resume();
    // Displays the choice
    commandTextArea.appendText("resume Button pressed !" + "\n");
```

```
    System.out.println("resume Button pressed !" + "\n");
  } // end action performed function
}

class insert_COM_NumberListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Sends  the inserted serial port number to the serial reader
    SimpleRead.comNumber = JOptionPane.showInputDialog(null,
        "Insert the com number for the serial port (COM#)");
  } // end action performed function
}

class straghtScenarioListener
    extends Frame
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Sends to the motionDetector the scenario choice
    motDetector.setScenario("staight");
    // Display instructions
    dataTextArea.appendText("\n" + "\n" +
        "**********************************************************” + “\n”
        "Your Choice is the Straigh Road Scenario." +
        "\n" + "\n" +

        "To Input your node's configuration " +
        "insert them in the Left window following the " +
        "instructions below:" + "\n" + "\n" +
        "The maximun number of nodes that the system can handle is 8." + "\n" +
        "Insert the node's data from the furthest to the nearest " +
        "following the order that the figure presents."  + "\n" +
        "Insert the values based on the following the format:" + "\n" +
        "\n" +
```

```
        "          nodeid: , distance from the previous node: , distance from the next node: "
        + "\n" +
        "         e.g. 0,2,2 " + "\n" + "\n" +
        "When you will finish type -end- and choose the 'Custom' " +
        "choice under the 'Configure' menu option.");

    System.out.println("staight choice!" + "\n");
    // get the image from a spevific file
    getImage("straight-road_small.jpg");
  } // end action performed function
} // end inner class straghtScenarioListener

class TScenarioListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Sends to the motionDetector the scenario choice
    motDetector.setScenario("TRoad");
    // Display instructions
    dataTextArea.appendText("\n" + "\n" +
        "**********************************************************" + "\n" +
        "Your Choice is the T Road Scenario." +  "\n" + "\n" +
        "To Input your node's configuration " +
        "insert them in the Left window following the " +
        "instructions below:" + "\n" + "\n" +
        "The maximun number of nodes that the system can handle is 8." + "\n" +
        "Insert the node's data following the order that the figure presents."  + "\n" +
        "Insert the values based on the following the format:" + "\n" +  "\n" +
        "          nodeid: , distance from the previous node: , distance from the next node: "
        + "\n" +
        "         e.g. 0,2,2 " + "\n" + "\n" +
        "When you will finish type -end- and choose the 'Custom' " +
        "choice under the 'Configure' menu option.");
```

```
    System.out.println("T-road choise!" + "\n");
    // get the image from a spevific file
    getImage("T-road_small.jpg");
  } // end action performed function
} // end inner class TScenarioListener

class crossroadsScenarioListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Sends to the motionDetector the scenario choice
    motDetector.setScenario("crossroads");
    // Display instructions
    dataTextArea.appendText("\n" + "\n" +
        "************************************************************" + "\n" +
        "Your Choice is the crossroads Scenario." +  "\n" + "\n" +
        "To Input your node's configuration " +
        "insert them in the Left window following the " +
        "instructions below:" + "\n" + "\n" +
        "The maximun number of nodes that the system can handle is 8." + "\n" +
        "Insert the node's data from the furthest to the nearest " +
        "following the order that the figure presents."  + "\n" +
        "Insert the values based on the following the format:" + "\n" + "\n" +
        "          nodeid: , distance from the previous node: , distance from the next node: "
        + "\n" +
        "         e.g. 0,2,2 " + "\n" + "\n" +
        "When you will finish type -end- and choose the 'Custom' " +
        "choice under the 'Configure' menu option.");

    System.out.println("crossroads choise!" + "\n");
    // get the image from a spevific file
    getImage("crossroads_small.jpg");
  } // end action performed function
```

```
} // end inner class crossroadsScenarioListener

class defaultListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // It reads the configuration file
    // It sends the nodes' characteristics to the motionDetector
    // to prepare the set up
    readFile("object_tracking.txt");
  } // end action performed function
} // end inner class defaultListener

class customListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {

    String[] tempStringArray = new String[8];
    String test = commandTextArea.getText();
    StringTokenizer st = new StringTokenizer(test, "\n");
    int i = 0;
    int j = 0;
    int k = 0;
    while (st.hasMoreTokens()) {
      tempStringArray[i] = st.nextToken();
      System.out.println(tempStringArray[i]);
      StringTokenizer st2 = new StringTokenizer(tempStringArray[i], ",");
      while (st2.hasMoreTokens()) {
        userInputs[j][k] = Integer.parseInt(st2.nextToken());
        k++;
      }
      k = 0;
      i++;
      j++;
```

```
    }

    while (j < 8) {
      for (int w = 0; w < 3; w++) {
        userInputs[j][w] = 0;
      }
      j++;
    }

    // sends the nodes' characteristics to the motionDetector
    motDetector.initializeNodes(userInputs);
  } // end action performed function
} // end inner class insertListener

private File imageFile;
private String localDirectory = "/j2sdk1.4.1_02/bin/configuration_file";
private FileInputStream fis = null;
private static byte[] imageByteArray = null;
private Image image = null;

/**
 *
 * @param name
 */
private void getImage(String name) {
  try {
    // read the file into a byte array
    imageFile = new File(localDirectory + "\\" + name);
    int length = 0;
    if (imageFile.exists()) {
      try {
        fis = new FileInputStream(imageFile);
        length = fis.available();
```

```
    }
    catch (Exception e) {
      System.out.println(
          "PROBLEM WITH fis = new FileInputStream(imageFile);");
    }

    imageByteArray = new byte[length];
    try {
      fis.read(imageByteArray);
    }
    catch (Exception e) {
      System.out.println("PROBLEM WITH fis.read(imageByteArray)");
    }

    //get an Image object from the payload bitstream
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    image = toolkit.createImage(imageByteArray, 0, length);

    //display the image as an ImageIcon object
    icon = new ImageIcon(image);
    iconLabel.setIcon(icon);
  }
}
catch (Exception e) {
  System.out.println(" Problem With the getImage method" + e);
}
finally {
  try {
    if (fis != null) {
      fis.close();
    }
  }
  catch (Exception e) {}
```

```
  }
} // End getImage

/**
 *
 * @param name
 */
private void readFile(String name) {
  String scenario = null;
  String comPort = null;
  String nodeid = null;
  String hold;
  String temp;
  int i = 0;
  int j = 0;

  try {
    FileReader fr = new FileReader(localDirectory + "\\" + name);
    BufferedReader br = new BufferedReader(fr);

    try {
      while ( (hold = br.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(hold, ":");
        temp = st.nextToken();
        if (temp.startsWith("scenario")) {
          scenario = st.nextToken();
          System.out.println("Scenario from file: " + scenario);
        }
        else if (temp.startsWith("comPort")) {
          comPort = st.nextToken();
          System.out.println("comPort from file: " + comPort);
        }
        else if (temp.startsWith("nodeid") && (i < 8) && (j < 3)) {
```

```
          userInputs[i][j] = Integer.parseInt(st.nextToken());
          System.out.println("for system id " + i + " nodeid from file: " +
                          userInputs[i][j]);
          j++;
        }
        else if (temp.startsWith("preDistance") && (i < 8) && (j < 3)) {
          userInputs[i][j] = Integer.parseInt(st.nextToken());
          System.out.println("preDistance:  from file: " + userInputs[i][j]);
          j++;
        }
        else if (temp.startsWith("postDistance") && (i < 8) && (j < 3)) {
          userInputs[i][j] = Integer.parseInt(st.nextToken());
          System.out.println("postDistance  from file: " + userInputs[i][j]);
          i++;
          j = 0;
        }
      }
      fr.close();
    }
    catch (IOException ex1) {
    }
  }
  catch (FileNotFoundException ex) {
  }

  // sets the COM#
  SimpleRead.comNumber = comPort;
  // sets the scenario
   motDetector.setScenario(scenario);
  // sends the nodes' characteristics
  motDetector.initializeNodes(userInputs);
}
```

```
 private void setDefaultNodeData() {

   for (int i = 0; i < 8; i++) {
     int j = 0;
     userInputs[i][0] = j;
     for (j = 1; j < 3; j++) {
       userInputs[i][j] = 10;
     }
   }
 }

 private void jbInit() throws Exception {
 } // end inner class pauseButtonListener

}



/**
 * <p>Title: node </p>
 * <p>Description: This is a utility class. Except the specific system that
 *               it is facilitate with or without major changes it can be used
 *               with any other similar hardware or topology.
 *               The purpose of this class is to hold data related
 *               to the ID and topological characteristic of the Crossbows
 *               MSP410 system nodes.
 *               Each node object carries: The distances from its neighbor
 *               nodes. The node ID as it is predefined in the node's software
 *               (normally it is the same ID with the one at the top of the
 *               Crossbows MSP410 node's protection case. Additional it holds
 *               the system ID. System ID is an internal ID for the specific
 *               system and depends on the current topological scenario that
 *               the Tracking Object application runs. Finally, the object
```

```
 *                holds the distances from the current node to the camera.
 *                </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

public class node {
  public int nodeId = 0;
  public int systemId = 0;
  public int preDistance = 20;
  public int postDistance = 20;
  public int distanceToCamera = 0;
  public int distanceLastNodeCamera = 5;

  public node() {
  }
  public void setNodeId(int inputNodeId){
    nodeId = inputNodeId;
  }
  public void setSystemId(int inputSystemId){
    systemId = inputSystemId;
  }
  public void setPreDistance(int intupPreDistance){
    preDistance = intupPreDistance;
  }
  public void setPostDistance(int intupPostDistance){
    postDistance = intupPostDistance;
  }
  public void setDistanceToCamera(int inputDistanceToCamera){
    distanceToCamera = inputDistanceToCamera;
  }
```

```
}
/**
 *
 * <p>Title: SimpleRead </p>
 * <p>Description: This class is used as a Serial Port reader for the Object
 *              Tracking application. It opens a serial port connection
 *              and a related input stream, and it read the raw data (bytes)
 *              returned from the Crossbow wireless sensor network system
 *              MSP410. Then it extract the useful information based on the
 *              message format that the raw data have and place them
 *              inside an array in order to be available from the rest
 *              software components of the application. The part of this class
 *              that reads data from the serial port is based
 *              on the SimpleRead.java file provided from the following URL
 *         ref: java.sun.com/developer/ releases/javacomm/SimpleRead.java May 2005
 *              </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

import java.io.*;
import java.util.*;
import javax.comm.*;
import javax.swing.Timer;

public class SimpleRead implements Runnable, SerialPortEventListener {

  static CommPortIdentifier portId;
  static Enumeration portList;
  InputStream inputStream;
  SerialPort serialPort;
```

```
static Thread readThread;
int numBytes;
static String comNumber = "COM5";
int counter;
// buffer to store the incomming messages
byte[] holdArray = new byte[1000];
//array used to store and send the proper data
int[] dataArray = new int[8];
//flag to control the program
static boolean flag = true;

/**
 * <p>Title: begin </p>
 * <p>Description: it open the com port and start reading data
 */
public static void begin() {
  portList = CommPortIdentifier.getPortIdentifiers();
  flag = true;

  while (portList.hasMoreElements()) {
    portId = (CommPortIdentifier) portList.nextElement();
    if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
      if (portId.getName().equals(comNumber)) {
        SimpleRead reader = new SimpleRead();
      }
    }
  }
}

public static void pause(){
  flag = false;
}
```

```
public static void resume(){
  flag = true;
}

public static void exit(){
  //exit the system
  System.exit(0);
}

/**
 * Constructor declaration
 */
public SimpleRead() {
  try {
    serialPort = (SerialPort) portId.open("SimpleReadApp", 20000);
  } catch (PortInUseException e) {System.out.println("port in use!!!!!!");}
  try {
    inputStream = serialPort.getInputStream();
  } catch (IOException e) {}
  try {
    serialPort.addEventListener(this);
  } catch (TooManyListenersException e) {}
  serialPort.notifyOnDataAvailable(true);
  try {
    serialPort.setSerialPortParams(57600,
                                   SerialPort.DATABITS_8,
                                   SerialPort.STOPBITS_1,
                                   SerialPort.PARITY_NONE);

    serialPort.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);

  } catch (UnsupportedCommOperationException e) {}
  readThread = new Thread(this);
```

```
  readThread.start();
}

public void run() {
  while ( flag == true) {
    try {
      Thread.sleep(2000);
    }
    catch (InterruptedException e) {}
  }
}

/**
 * Title: serialEvent
 * Description: it sets the serial port parameters and places the data
 *              in a buffer
 * @param event
 */
public void serialEvent(SerialPortEvent event) {
  switch(event.getEventType()) {
    case SerialPortEvent.BI:
    case SerialPortEvent.OE:
    case SerialPortEvent.FE:
    case SerialPortEvent.PE:
    case SerialPortEvent.CD:
    case SerialPortEvent.CTS:
    case SerialPortEvent.DSR:
    case SerialPortEvent.RI:
    case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
      break;
    case SerialPortEvent.DATA_AVAILABLE:
      try {
        while (inputStream.available() > 0 && flag == true) {
```

```
        byte[] readBuffer = new byte[41];
        numBytes = inputStream.read(readBuffer);
        // Passes the data that the buffer holds into the message method
        message(readBuffer);
       }
     } catch (IOException e) {}
     break;
  }
}

/**
 * <p>Title: message </p>
 * <p>Description: This method first reconstract the message that the MSP410
 *              nodes send through the gateway to the serial port. When the
 *              message is completed it reads and places the importand
 *              values to an array and passes them to the rest software
 *              components of the Tracking Object application
 *
 * @param readBuffer byte[]
 */
private void message(byte[] readBuffer){
  // data variables
  int SeqNumber;
  int SeqNumberF;
  int vref;
  int quad;
  int pir;
  int mag;
  int audio;
  // it initializes a motionDetector object that it is used later to pass the
  // the data to the motionDetector class.
  motionDetector detector = new motionDetector();
```

```
// reconstracts the message
if ( readBuffer[0] == 126 ){
  counter = 0;
  for( int i = 0; i < numBytes; i++){
    holdArray[counter] = readBuffer[i];
    if (i >= 28 && readBuffer[i] == 126) {
      counter = 0;
    }
    counter++;
  }
}
// it stores the data into the data array
else{
  for( int k = 0; k < numBytes; k++){
    holdArray[counter] = readBuffer[k];
    if (counter == 38 && readBuffer[k] == 126) {
      //store the nodeid
      dataArray[0] = unsigned_int(holdArray[11]);
      //store the parentid
      dataArray[1] = unsigned_int(holdArray[19]);
      //calcutate the seq# and store it
      SeqNumber = unsigned_int(holdArray[13]) +
          256 * unsigned_int(holdArray[14]);
      SeqNumberF = unsigned_int(holdArray[20]) +
          256 * unsigned_int(holdArray[21]);
      dataArray[2] = SeqNumberF;
      //calcutate the vref and store it
      vref = unsigned_int(holdArray[22]);
      dataArray[3] = vref;
      //calcutate the quad1 and store it
      quad = unsigned_int(holdArray[23]);
      dataArray[4] = quad;
      //calcutate the pir and store it
```

```
        pir = unsigned_int(holdArray[24]) +
            256 * unsigned_int(holdArray[25] & 0x03);
        dataArray[5] = pir;
        //calcutate the mag and store it
        mag = unsigned_int(holdArray[26]) +
            256 * unsigned_int(holdArray[27] & 0x03);
        dataArray[6] = mag;
        //calcutate the audio and store it
        audio = unsigned_int(holdArray[28]) +
            256 * unsigned_int(holdArray[29] & 0x03);
        dataArray[7] = audio;
        // send the data to the motionDetector class for furture manipulation
        detector.receiveData(dataArray);

        //test code print the received values in a command linr window
        System.out.println("--------------------RAW DATA--------------------" + "\n"+
                           "  id: " + dataArray[0] + "  parent: " + dataArray[1] +
                           " seq#FINAL: " + dataArray[2] +
                           " vref: " + dataArray[3] +
                           " quad: " + dataArray[4] + " pir: " + dataArray[5] +
                           " mag: " + dataArray[6] + " audio: " + dataArray[7] + "\n" +
                           "-----------------------------------------------------");
        counter = 0;
      }
      counter++;
    }
  }
}

/**
 * Description: It convert the received integer value to unsigned int
 * @param nb
 * @return
```

```
  */
  static int unsigned_int(int nb){
    if(nb >= 0 )
      return nb;
    else
      return(256+nb);
  }
}
```

```
/**
 * <p>Title:       motionDetector </p>
 * <p>Description: This class is responsible to implement the first part of the
 *                 part of the object-tracking application algorithm.
 *                 it is responsible first to reseive the data from the SimpleRead
 *                 class, then it determines if the received data are related
 *                 to an object's detection.
 *                 If the system has a detection it determines the object id
 *                 and then it stores all the information to a target object.
 *                 The target object then is forwarded to the proper scenario
 *                 class for further process.
 *                 Additionally, the class maintains and updates the proper
 *                 pir and mag thresholds.
 *                 Finally, the motionDetector class contains utility method
 *                 for the scenarios classes. Those methods are responsible
 *                 to control the systme's nodes data structures that hold the
 *                 data. Moreover they control the print out messages and trigger
 *                 whenever required the TSSRv3 system.
 *
 * Assumptions:    The objects inside the system do not change order.
```

```
 *              Inside the application exist up to 3 objects per time with the
 *              same direction.
 *              Inside the application exist objects with te same direction.
 *              </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

import java.io.File;
import java.text.NumberFormat;

public class motionDetector {
  // variables to hold the data
  // the commented variables are for future use
  public int nodeid;
  //private int parent;
  public int systemId;
  public double currentTime;
  public int seqNumber;
  //private int voltage;
  public int quad;
  public int pir;
  //private int audio;
  public int mag;
  public double speed;
  public String incomingObject;
  public String direction;
  public double timeToCamera;

  // variables used for the thresholds
  public static int magneticThr;
```

```
public static int irThr;

// file names used to triger the TSSRv3 system
public static File file1, file2;

// variables used as indexes in the
public static int MAX_CAR_ARRAY_IDEX;
public static int MAX_HUMAN_ARRAY_IDEX;
public static int MAX_THRESH_ARRAY_INDEX = 20;

// Initiate a double array to hild the received and produced object's data
public static motionDetector[][] CarQueue;
// the counter for the above queue
public static int[] countersQueue;

//  variable that holds the senario that the program will use
private static String Senario;

// store the node's physical characteristics
private static node[] nodeArray = new node[8];

// Constract a new object for straight road scenario
// in order to call the proper methods later
private straightRoadScenario SRScenario = new straightRoadScenario();
// Constract a new object for T road scenario
// in order to call the proper methods later
private TRoadScenario TRScenario = new TRoadScenario();
// Constract a new object for crossroads  scenario
// in order to call the proper methods later
private crossroadsScenario CRScenario = new crossroadsScenario();

// utility variables to store the speed history
public double[] speedHistory;
```

```
public int speedHistorySize = 8;

/**
 * Constructor
 */
public motionDetector() {
}

public static void setScenario(String scenario){
  Senario = scenario;

}
/**
 * Title: initialize
 * Description: Initialize the counters
 */
public static void initialize(){
  // set the system's properties
  // initialize the counters for each node array
  countersQueue[0] = 0;
  countersQueue[1] = 0;
  countersQueue[2] = 0;
  countersQueue[3] = 0;
  countersQueue[4] = 0;
  countersQueue[5] = 0;
  countersQueue[6] = 0;
  countersQueue[7] = 0;
} // end initialize

/**
 * Title: initializeNodes
 * Description: Initialize the system nodes
 * @param senario
```

```
 * @return
 */
public void initializeNodes(int[][] userInputs){
  setThresholds();
  setNodesID(userInputs);
  nodeArray = setDefaultDistances(Senario, nodeArray, userInputs);
}

/**
 * Title: setNodesID
 * Description: Build node objects and determines the system's id that corresponds
 *              to the physical node id and stores the data in the node[] array.
 *              The physical node id is containe in the userInput double array
 * @param userInputs
 */
public void setNodesID(int[][] userInputs){

  node node1 = new node();
  node1.setSystemId(0);
  node1.setNodeId(userInputs[0][0]);
  nodeArray[0] = node1;
  node node2 = new node();
  node2.setSystemId(1);
  node2.setNodeId(userInputs[1][0]);
  nodeArray[1] = node2;
  node node3 = new node();
  node3.setSystemId(2);
  node3.setNodeId(userInputs[2][0]);
  nodeArray[2] = node3;
  node node4 = new node();
  node4.setSystemId(3);
  node4.setNodeId(userInputs[3][0]);
  nodeArray[3] = node4;
```

```
  node node5 = new node();
  node5.setSystemId(4);
  node5.setNodeId(userInputs[4][0]);
  nodeArray[4] = node5;
  node node6 = new node();
  node6.setSystemId(5);
  node6.setNodeId(userInputs[5][0]);
  nodeArray[5] = node6;
  node node7 = new node();
  node7.setSystemId(6);
  node7.setNodeId(userInputs[6][0]);
  nodeArray[6] = node7;
  node node8 = new node();
  node8.setSystemId(7);
  node8.setNodeId(userInputs[7][0]);
  nodeArray[7] = node8;
}

/**
 * Title: setDefaultDistances
 * Description: Based on the scenario it forwareds the user's imput to the
 *              proper scenario class in order for the insert distances to be set
 */
public node[] setDefaultDistances(String senario, node[] initialNodeArray,
                                  int[][] userInputs){
  if ( senario == "staight" ){
    straightRoadScenario CSRScenario = new straightRoadScenario();
    SRScenario.setDefaultDistances(initialNodeArray, userInputs);
  }
  else if ( senario == "TRoad" ){
    straightRoadScenario CSRScenario = new straightRoadScenario();
    TRScenario.setDefaultDistances(initialNodeArray, userInputs);
  }
```

```
  else if ( senario == "crossroads" ){
    straightRoadScenario CSRScenario = new straightRoadScenario();
    CRScenario.setDefaultDistances(initialNodeArray, userInputs);
  }

  return initialNodeArray;
} // End setDefaultDistances

/**
 * Title: setThresholds
 * Description: Intitializes the threshold values and the arrays
 */
public void setThresholds(){
  // initialize the default file
  motionDetector.file1 = new File("C:/LOADSHARE/x.txt");
  // set the thresholds
  magneticThr = 300;
  irThr = 700;
  // the value at this threshold is static and it is
  // compute for speed 10Km/h (2.777m/sec)
  // it is used to reset the system if it has a lot of time to detect an object
  timeThreshold = 15000;
  // set the array index for the objects
  MAX_CAR_ARRAY_IDEX = 4;
  // initialize the FIFO data structure for the detected objects
  CarQueue = new motionDetector[8][MAX_CAR_ARRAY_IDEX];
  // initialize an array that holds the counetrs for the queues
  countersQueue = new int[8];
} // End setThresholds

/**
 * Title: receiveData
 * Description: This is the first step of the algorithm.
```

```
*           It receives the data from the SimpleRead class and determines
*           if the returned message is a detection message or not.
*           It also, based on the returned values determines if the
*           detected object is car or human.
*           Then it places the reseved data and the default values for the
*           motion tracking into an object called target.
*           Then, based on the selected from the user scenario
*           it forwards the target object to the proper scenario class.
*           Finally, it is responsible to update the threshold values that
*           the application uses.
* @param data
*/
public void receiveData(int [] data){
 // build a target object to store the received and produsedd data
 motionDetector target = new motionDetector();

 // it stores the system time into the target object
 target.currentTime = System.currentTimeMillis();
 // The commented variables are for future use.
 // They are received but they are not currently used in this version of the
 // application.

 // Stores the received node id
 target.nodeid = data[0];
 //parent = data[1];
 // Stores the received message seqNumber
 target.seqNumber = data[2];
 //voltage = data[3];
 // Stores the received quad
 target.quad = data[4];
 // Stores the received pir
 target.pir = data[5];
 // Stores the received mag
```

```
target.mag = data[6];
//audio = data[7];
// Intialize and stores the default speed value
target.speed = 0;
// Intialize and stores the default incomingObject
target.incomingObject = "unknown";
// Intialize and stores the default direction
target.direction = "unkown";
// Intialize and stores the default timeToCamera
target.timeToCamera = 0;

// Test message that displays in the command line window the current
// values of the thresholds
System.out.println("********PirThr " + irThr + "MagThr " + magneticThr);

// The following block of selections are based on the selected
// from the user scenario.
// Inside the selections, it is implemented the first part of the algorithm
// that is responsible to determine the detection signals and the object's id.
// Then the update target object is forwarded in the proper scenario calss.
// Finally, if the message does not indicates a detection the data are used
// to update the thresholds
if ( Senario == "staight" ){
  // If the returned data contains only to the pir
  if( (( target.pir >= irThr) && (target.quad == 1 ))){
    // The incoming object is characterized as humman because the returned
    // values are related only to the pir
    target.incomingObject = "humman";
    // Initialize the speed history variables for the current target object
    target.speedHistory = new double[speedHistorySize];
    for (int i = 0; i < speedHistorySize; i++){
      target.speedHistory[i] = 0;
    }
```

```
  // It place the proper target id to the target object based on
  // the system ids
  target.systemId = setTargetId(target.nodeid, nodeArray);
  // Send the updated target object to the straight-road scebario
  // for additional process.
  SRScenario.detect(target, nodeArray);
}

// If the returned data contains both pir and mag
else if( (( target.pir >= irThr ) && (target.quad == 1 ))
    && (target.mag > target.magneticThr)){
  // The incoming object is characterized as car
  target.incomingObject = "car";
  // initialize the speed history variables for the current target object
  target.speedHistory = new double[speedHistorySize];
  for (int i = 0; i < speedHistorySize; i++){
    target.speedHistory[i] = 0;
  }
  // It place the proper target id to the target object based on
  // the system ids
  target.systemId = setTargetId(target.nodeid, nodeArray);
  // Send the updated target object to the straight-road scebario
  // for additional process.
  SRScenario.detect(target, nodeArray);
}

// If the reseved are not valid to produse a detection event, they are consider
// as environmental returns and they are used to update the application's
// thresholds.
else{
  // send the data to update the thresholds
  pirThreshold(data);
  magThreshold(data);
```

```
  }
}

// When the selected scenario is T-road the following selection block is run
// and implemetns the same stepts that the straight-road selection comments describe
else if ( Senario == "TRoad" ){
  if( (( target.pir >= irThr) && (target.quad == 1 ))){
    target.incomingObject = "humman";
    target.speedHistory = new double[speedHistorySize];
    for (int i = 0; i < speedHistorySize; i++){
      target.speedHistory[i] = 0;
    }
    target.systemId = setTargetId(target.nodeid, nodeArray);
    SRScenario.detect(target, nodeArray);
  }

  else if( (( target.pir >= irThr ) && (target.quad == 1 ))
      && (target.mag > target.magneticThr)){
    target.incomingObject = "car";
    target.speedHistory = new double[speedHistorySize];
    for (int i = 0; i < speedHistorySize; i++){
      target.speedHistory[i] = 0;
    }
    target.systemId = setTargetId(target.nodeid, nodeArray);
    SRScenario.detect(target, nodeArray);
  }

  else{
    pirThreshold(data);
    magThreshold(data);
  }
}
```

```
 // when it receives the data it call the detect of the crossroads senario
 // method in order to extract usefull information
 else if ( Senario == "crossroads" ){
  if( (( target.pir >= irThr) && (target.quad == 1 ))){
   target.incomingObject = "humman";
   target.speedHistory = new double[speedHistorySize];
   for (int i = 0; i < speedHistorySize; i++){
    target.speedHistory[i] = 0;
   }
   target.systemId = setTargetId(target.nodeid, nodeArray);
   SRScenario.detect(target, nodeArray);
  }

  else if( (( target.pir >= irThr ) && (target.quad == 1 ))
     && (target.mag > target.magneticThr)){
   target.incomingObject = "car";
   target.speedHistory = new double[speedHistorySize];
   for (int i = 0; i < speedHistorySize; i++){
    target.speedHistory[i] = 0;
   }
   target.systemId = setTargetId(target.nodeid, nodeArray);
   SRScenario.detect(target, nodeArray);
  }

  else{
   pirThreshold(data);
   magThreshold(data);
  }
 }
} //End receiveData

/**
 * Title: setTargetId
```

```
 * Description: Based on the insert node's topology from the user and the
 *              system id that the program produces. It checks the id of the
 *              node that send the detection and return the proper system id
 *              for the target objects that holds the data.
 * @param targetId
 * @param nodeArray
 * @return
 */
public int setTargetId( int targetId, node[] nodeArray){
  int systemID = 152;
  if (targetId == nodeArray[0].nodeId){
    systemID = 0;
  }
  else if (targetId == nodeArray[1].nodeId){
    systemID = 1;
  }
  else if (targetId == nodeArray[2].nodeId){
    systemID = 2;
  }
  else if (targetId == nodeArray[3].nodeId){
    systemID = 3;
  }
  else if (targetId == nodeArray[4].nodeId){
    systemID = 4;
  }
  else if (targetId == nodeArray[5].nodeId){
    systemID = 5;
  }
  else if (targetId == nodeArray[6].nodeId){
    systemID = 6;
  }
  else if (targetId == nodeArray[7].nodeId){
    systemID = 7;
```

```
  }

  return systemID;
}

/**
 * Title: resetCarArraysAndCounters
 * Description: Reset the counters and the node's arrays when the data that they
 *              hold are too old
 */
public void resetCarArraysAndCounters(){
  // reset the car counter
  countersQueue[0] = 0;
  countersQueue[1] = 0;
  countersQueue[2] = 0;
  countersQueue[3] = 0;
  countersQueue[4] = 0;
  countersQueue[5] = 0;
  countersQueue[6] = 0;
  countersQueue[7] = 0;

  // reset the car arrays
  CarQueue[0][0] = null;
  CarQueue[1][0] = null;
  CarQueue[2][0] = null;
  CarQueue[3][0] = null;
  CarQueue[4][0] = null;
  CarQueue[5][0] = null;
  CarQueue[6][0] = null;
  CarQueue[7][0] = null;
} // end resetCarArraysAndCounters

// utility variable to hold time and help to reset
```

```
// the arrays in order to have correct results
// it is used in the receiveData method
public static long oldTime = 0;

// valiable to set the time limit for the time check
// which is performed inside the receiveData method
// in order if the time space between the data is too
// long to reset the arrays.
public static long timeThreshold;

/**
 * Title: computeSpeed
 * Description: Compute the object's speed based on the time difference
 *              between the new and the old detection an in the distance
 *              between the nodes
 * @param time1
 * @param time2
 * @param distance
 * @return
 */
public double computeSpeed(double time1, double time2, int distance){
  double timeDiff = ((time2 - time1) / 1000);
  double speed = (distance /timeDiff ); // speed in m/sec
  return speed;
}

/**
 * Title: rearrangeArray
 * Description: rearange the node's array to free space
 *              by removing the oldest data
 * @param i
 * @param sysId
 * @param Counter
```

```
   * @param queue
   * @return
   */
  public motionDetector[][] rearrangeArray(int i, int sysId, int Counter,
                                  motionDetector[][] queue){
//    int i = 0;
    while ( i < Counter){
      queue[sysId][i] = queue[sysId][i + 1];
      i++;
    }
    // delete the last input of the array as reduntant
    queue[sysId][Counter] = null;
    return queue;
  }

  /**
   * Title: removeRedundantEntry
   * Description: Removes from the nodes' arrays old data that have already used
   *              and rearrange the arrays.
   *
   * @param seqNum
   */
  public void removeRedundantEntry(int seqNum){
    for (int i = 0; i < 8; i++){
     for (int j = 0; j < MAX_CAR_ARRAY_IDEX; j++){
       if (CarQueue[i][j] != null && CarQueue[i][j].seqNumber == seqNum){
         int k = j;
         while(k < MAX_CAR_ARRAY_IDEX - 1 ){
           CarQueue[i][k] = CarQueue[i][k + 1];
           k++;
         }
         CarQueue[i][MAX_CAR_ARRAY_IDEX - 1] = null;
       }
```

```
    }
  }
}

// Utility variables uses in the computation of the PIR threshold
private static int[] holdPirArray = new int[MAX_THRESH_ARRAY_INDEX];
private static int holdPirArrayCounter = 0;

/**
 * Title: pirThreshold
 * Description: Receives the returned values from the sensors
 *              that are not related to the detections and calculates the
 *              PIR threshold.
 * @param rawData
 */
private void pirThreshold( int[] rawData){
  holdPirArray[holdPirArrayCounter] = rawData[5];
  if ( holdPirArrayCounter == MAX_THRESH_ARRAY_INDEX - 1){
    int sum = 0;
    for (int i = 0; i < MAX_THRESH_ARRAY_INDEX; i++){
      sum = sum + holdPirArray[i];
    }
    irThr = sum / MAX_THRESH_ARRAY_INDEX;
    holdPirArrayCounter--;
  }
  else{
    holdPirArrayCounter++;
  }
}
// Utility variables uses in the computation of the mag threshold
private static int[] holdMagArray = new int[MAX_THRESH_ARRAY_INDEX];
private static int holdMagArrayCounter = 0;
```

```
/**
 * Title: magThreshold
 * Description: Receives the returned values from the sensors
 *              that are not related to the detections and calculates the
 *              mag threshold.

 * @param rawData
 */
private void magThreshold( int[] rawData){
  holdMagArray[holdMagArrayCounter] = rawData[6];
  holdMagArrayCounter++;
  if ( holdMagArrayCounter == MAX_THRESH_ARRAY_INDEX - 1){
    int sum = 0;
    for (int i = 0; i < MAX_THRESH_ARRAY_INDEX; i++){
      sum = sum + holdMagArray[i];
    }
    magneticThr = sum / MAX_THRESH_ARRAY_INDEX;
    holdMagArrayCounter--;
  }
}

/**
 * Title: computeTimeToCamera
 * Description: Based on the object's distance to camera and speed it
 *              computes the estimated arrival time to the camera's
 *              focal point
 * @param distance
 * @param speed
 * @return
 */
public double computeTimeToCamera(int distanceToCamera, double speed){
  double timeToCamera = distanceToCamera / speed;
  return timeToCamera;
```

```
}

// utility variable for the sendCommand in order
// to manage appropriate the time delays
private static int oldNodeid = 0;
private static double oldSpeed = 0;
private static motionDetectionSystem MDSystem = new motionDetectionSystem();

/**
 * Title: sendCommand
 * Description: It responsible to inform the user with the produsedata
 *              and to trigger the TSSRv3 system whenever is important
 *
 * @param nodeid
 * @param incomingObject
 * @param direction
 * @param ditectionTime
 * @param timeToCamera
 */
public void sendCommand(int sysId, String incomingObject,
                        String direction, double speed,
                        double timeToCamera, double[] speedHistory){

  // Prints the produsedata in the application's command line window
  System.out.println("**************************************************");
System.out.println( "sysId: " + sysId  + "  incomingObject: " + incomingObject +
                    "  direction: " + direction + " speed: " + speed +
                    "  timeToCamera: " + timeToCamera);

System.out.println("**************************************************");
  // Displays the produsedata in the application's GUI
  MDSystem.printData( "sysId: " + sysId  +
                              "  incomingObject: " +incomingObject +
```

```
                        "  direction: " + direction +
                        "  speed: " + speed +
                        "  timeToCamera: " + timeToCamera +
                        "\n");

    // This selsction triggers the TSSRv3 system and outputs the data
    // when the object passes the closest node to the base station
    // it an additional trigger to the TSSRv3 system that increases the system's
    // reduntancy
    // for deploymend that involves 8 nodes use this selection block
    if ( sysId == 7 ){
      // it prints in the proper text area the comands
      changeFileName();
      MDSystem.printCommands(" TAKE PICTURE OBJECT NEAR NODE " + sysId +
"\n");
      MDSystem.printData( "sysId: " + sysId  +
                        "  incomingObject: " +incomingObject +
                        "  direction: " + direction +
                        "  speed: " + speed +
                        "  timeToCamera: " + timeToCamera +
                        "\n");
      // it trigers the TSSRv3 system by calling the changeFileName
      // which renames the file name (file name is the trigger)
      changeFileName();
    }

    // for deploymend that involves 6 nodes commend the above selection block
    // and uncommend and use this
/*
    if ( sysId == 5 ){
      // it prints in the proper text area the comands
      changeFileName();
```

```
    MDSystem.printCommands(" TAKE PICTURE OBJECT NEAR NODE " + sysId +
"\n");
    MDSystem.printData( "sysId: " + sysId  +
                        "  incomingObject: " +incomingObject +
                        "  direction: " + direction +
                        "  speed: " + speed +
                        "  timeToCamera: " + timeToCamera +
                        "\n");
    // it trigers the TSSRv3 system by calling the changeFileName
    // which renames the file name (file name is the trigger)
    changeFileName();
   }
*/
   // For all the incoming objects
   if( direction.startsWith("inbound") ){
    MDSystem.printCommands("\n" + " INCOMING OBJECT  " + "\n");
    MDSystem.printData( "*******************************************" +
                "sysId: " + sysId  +
                "  incomingObject: " +incomingObject +
                "  direction: " + direction +
                "  speed: " + speed +
                "  timeToCamera: " + timeToCamera +
                "*************************************************" +
                "\n");
    // It uses the waitTime class to produse the proper delay to trigger the
    // TSSRv3 system
    // The delay is equal to the estimated arrival time (timeToCamera)
    if (waitTime.flag == true){
     waitTime.StopTask();
     // initiate a new delay trhead
     waitTime newDelay = null;
     // starts the thread
     newDelay.StartTask( (int) timeToCamera);
```

```
    }

    else{
      // initiate a new delay trhead
      waitTime newDelay = null;
      // starts the thread
      newDelay.StartTask( (int) timeToCamera);
    }
  }
  // call the printSpeedHistory to print out the speed history
  printSpeedHisrory( speedHistory );

} // End sendCommand

/**
 * Title: printSpeedHisrory
 * Description: Print out into the comand line window and in the GUI the
 *              object's speed history and the average speed
 * @param counter
 * @param array
 */
public void printSpeedHisrory(double[] array ){
  double sum = 0;
  int j = 0;
  for (int i = 0; i < speedHistorySize; i++){
    if ( array[i] != 0){
      System.out.println(" The object in " + i + " node had Speed " + array[i]);
      MDSystem.printData(" The object in " + i
                    + " node had Speed "
                    + array[i]
                    + "\n");
      j++;
      sum = sum + array[i];
```

```
    }
   }
   System.out.println(" The Average Speed was " + sum/j);
  }

  /**
   * Title: printSpeedHisrory
   * Description: Change the file name in order to triger the TSSRv3 system
   *
   */
  public void changeFileName(){
   // alll the files must be in the C:/LOADSHARE/ directory
   // the TSSRv3 system uses the file name x.txt and the object-tracking
   // application (this program) the 1.txt
   file2 = new File("C:/LOADSHARE/1.txt");
   file1.renameTo(file2);
  }
}




/**
 * <p>Title: waitTime </p>
 * <p>Description: This class is used as time Tread for the Tracking Object
 *              application. It is uses the java.util.Timer to schedule a
 *              task. This task is a time delay for the TSSRv3 system in order
 *              to delay the camera's trigger proper time. It is also able to
 *              start and stop the timer whenever a new more update data are
 *              available.
 *              This class is based in the a similar class provided by
 *              sun tutorials.
```

```
 *:http://java.sun.com/docs/books/tutorial/essential/threads/timer.html 06/06/2005
 *                </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

import java.util.Timer;
import java.util.TimerTask;

public class waitTime {
  static Timer timer;
  static boolean flag;
  static motionDetectionSystem MDSystem = new motionDetectionSystem();
  static motionDetector detector = new motionDetector();

  /**
   * Initialize the timer and mke the conversion from seconds to miliseconds
   * @param seconds
   */
  public waitTime(long seconds) {
    timer = new Timer();
    timer.schedule(new RemindTask(), seconds*1000);
  }

  /**
   * starts the countdown and trigger the TSSRv3 camera
   *
   */
  class RemindTask extends TimerTask {

    public void run() {
```

```
    System.out.println("Time's up!  ************************");
    timer.cancel(); //Terminate the timer thread
    MDSystem.printCommands(" THE CAMERA FIRES" + "\n");
    // change the file name to triger the camera
    detector.changeFileName();
    flag = false;
   }
  }

  /**
   * Initiate a task
   * @param timeToCamera
   */
  public static void StartTask(long timeToCamera) {
   flag = true;
   new waitTime(timeToCamera);
   MDSystem.printCommands(" THE COUNTDOWN STARTS" + "\n");
   System.out.println("Task scheduled.");
  }

  /**
   * It stops the task
   */
  public static void StopTask(){
   System.out.println("Time stopped!");
   MDSystem.printCommands(" THE COUNTDOWN STOPS" + "\n");
   timer.cancel(); //Terminate the timer thread
   flag = false;
  }
 }
```

```
/**
 * <p>Title: straightRoadScenario</p>
 * <p>Description: This class is responsible to implement the algorithmic
 *              process for the straight-road scenario by receiving the
 *              data from the motionDetector class.
 *              First based on the raw data and in the stored data it produces
 *              the direction of the object, and then the speed.
 *              Finally, it is responsible to inform the data structures of
 *              the neighbor nodes to continue tracking the object.</p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

import java.io.File;

public class straightRoadScenario {
  public straightRoadScenario() {
  }
  // Initialize a motionDetector object in order to be able to call the
  // proper methods
  private static motionDetector detector = new motionDetector();

  /**
   * Title: setDefaultDistances
   * Description: It takes the data related to the nodes physical characteristics
   *              and place them in a array of node objects.
   * @param initialNodeArray
   * @param userInputs
   * @return
```

```
*/
public node[] setDefaultDistances(node[] initialNodeArray ,
                                  int[][] userInputs){
  //set the distances in the array that holds the data related to nodes
 initialNodeArray[0].setPreDistance(userInputs[0][1]);
 initialNodeArray[0].setPostDistance(userInputs[0][2]);
 initialNodeArray[1].setPreDistance(userInputs[1][1]);
 initialNodeArray[1].setPostDistance(userInputs[1][2]);
 initialNodeArray[2].setPreDistance(userInputs[2][1]);
 initialNodeArray[2].setPostDistance(userInputs[2][2]);
 initialNodeArray[3].setPreDistance(userInputs[3][1]);
 initialNodeArray[3].setPostDistance(userInputs[3][2]);
 initialNodeArray[4].setPreDistance(userInputs[4][1]);
 initialNodeArray[4].setPostDistance(userInputs[4][2]);
 initialNodeArray[5].setPreDistance(userInputs[5][1]);
 initialNodeArray[5].setPostDistance(userInputs[5][2]);
 initialNodeArray[6].setPreDistance(userInputs[6][1]);
 initialNodeArray[6].setPostDistance(userInputs[6][2]);
 initialNodeArray[7].setPreDistance(userInputs[7][1]);
 initialNodeArray[7].setPostDistance(userInputs[7][2]);

 initialNodeArray[0].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[0].systemId, initialNodeArray));
 initialNodeArray[1].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[1].systemId, initialNodeArray));
 initialNodeArray[2].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[2].systemId, initialNodeArray));
 initialNodeArray[3].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[3].systemId, initialNodeArray));
 initialNodeArray[4].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[4].systemId, initialNodeArray));
 initialNodeArray[5].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[5].systemId, initialNodeArray));
```

```
 initialNodeArray[6].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[6].systemId, initialNodeArray));
 initialNodeArray[7].setDistanceToCamera(
    setSRSNodeDistanceToCamera(initialNodeArray[7].systemId, initialNodeArray));
 return initialNodeArray;
}

/**
 * Title: setSRSNodeDistanceToCamera
 * Description: It calculates the distance from the node to the
 *              TSSRv3 camera-base station
 * @param systemId
 * @param initialNodeArray
 * @return
 */
public int setSRSNodeDistanceToCamera(int systemId, node[] initialNodeArray){
 int distanceToCamera = 0;
 if (systemId == 0){
   distanceToCamera = initialNodeArray[0].postDistance +
                      initialNodeArray[1].postDistance +
                      initialNodeArray[2].postDistance +
                      initialNodeArray[3].postDistance +
                      initialNodeArray[4].postDistance +
                      initialNodeArray[5].postDistance +
                      initialNodeArray[6].postDistance +
                      initialNodeArray[7].distanceLastNodeCamera;
 }
 else if (systemId == 1){
   distanceToCamera = initialNodeArray[1].postDistance +
                      initialNodeArray[2].postDistance +
                      initialNodeArray[3].postDistance +
                      initialNodeArray[4].postDistance +
                      initialNodeArray[5].postDistance +
```

```
                initialNodeArray[6].postDistance +
                initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 2){
  distanceToCamera = initialNodeArray[2].postDistance +
                initialNodeArray[3].postDistance +
                initialNodeArray[4].postDistance +
                initialNodeArray[5].postDistance +
                initialNodeArray[6].postDistance +
                initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 3){
  distanceToCamera = initialNodeArray[3].postDistance +
                initialNodeArray[4].postDistance +
                initialNodeArray[5].postDistance +
                initialNodeArray[6].postDistance +
                initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 4){
  distanceToCamera = initialNodeArray[4].postDistance +
                initialNodeArray[5].postDistance +
                initialNodeArray[6].postDistance +
                initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 5){
  distanceToCamera = initialNodeArray[5].postDistance +
                initialNodeArray[6].postDistance +
                initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 6){
  distanceToCamera = initialNodeArray[6].postDistance +
                initialNodeArray[7].distanceLastNodeCamera;
}
```

```
  else if (systemId == 7){
    distanceToCamera = initialNodeArray[7].distanceLastNodeCamera;
  }
  else {
    System.out.println(" Node Input Error ");
  }
  return distanceToCamera;
}

/**
 * Title: setDirection
 * Description: It implement the first part of the algorithm by producing the
 *              object's direction based on the node id that returns the
 *              detection and on the stored data in the node's data structure
 * @param systemId
 * @return
 */
public String setDirection( int sysId ){
  String direction = "unknown";

  if (detector.CarQueue[sysId][0] == null){
    direction = "unknown";
  }
  else if (sysId > detector.CarQueue[sysId][0].systemId ){
    direction = "inbound";
  }
  else if (sysId < detector.CarQueue[sysId][0].systemId ){
    direction = "outbound";
  }
  return direction;
}

/**
```

```
* Title: detect
* Description: It implement the second part of the algorithm.
*              By using the proper method's calls outputs the object's
*              direction, and speed and stored them in the target object
*              parameter.
*              Its main purpose it to keep the the FIFO uptadated. Thus,
*              by calling proper method's removes the old values. Aditioanlly
*              by evaluating the objects direction it updates the proper node's
*              data structures.
*
* The nodes topology that this scenario use based on the system id
* is the following
* The number represent the system id of the nodes
*
* --0---------2---------4---------6-----------Base Station (TSSRv3 Camera-----
*                 Road or Corridor
* -------1---------3---------5---------7-----------------------------------
*
* Or for one side deployment
* * --0----1----2----3----4----5----6---7--------Base Station (TSSRv3 Camera-----
*                 Road or Corridor
* ------------------------------------------------------------------------
*
* @param node
*/
public void detect(motionDetector target, node[] nodeArray){

  int distance = 0;

  // check if the stored data are too old
  if ( ( (long)target.currentTime - detector.oldTime ) >= detector.timeThreshold){
    // if the data are too old the motionDetector's resetCarArraysAndCounters()
    // method reset the counters and empty the arrays
```

```
 detector.resetCarArraysAndCounters();
}

// place the curent system time to the oldTime variable
// in order to check the next incoming message
detector.oldTime = (long)target.currentTime;

//Test code displays the stored thresholds in the command line window
System.out.println("PirThr" + detector.irThr + "MagThr" + detector.magneticThr);

// Call the setDirection to calculate the direction of the object
 target.direction = setDirection(target.systemId);

 // check if the node's data structure has stored target objects
 // If it has it use them  to calculate the object's speed
 if ( detector.CarQueue[target.systemId][0] != null ){
   // checks the direction of the object in order to use
   // the proper distance in the speed computation
   // if the direction is "unknown" it use s the default distance
   // place the proper distance if the stored value is from the neighbor node
   if (target.direction == "inbound"
       && (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1)){
    distance = nodeArray[target.systemId].preDistance;
   }

   // place the proper distance if the stored value is from the two node away
   else if (target.direction == "inbound"
       && (target.systemId - detector.CarQueue[target.systemId][0].systemId != 1)){
    distance = nodeArray[target.systemId].preDistance
        + nodeArray[detector.CarQueue[target.systemId][0].systemId].postDistance;
   }

   // place the proper distance if the stored value is from the neighbor node
```

```
  else if (target.direction == "outbound"
     && (detector.CarQueue[target.systemId][0].systemId - target.systemId == 1)){
   distance = nodeArray[target.systemId].postDistance;
  }

  // place the proper distance if the stored value is from the two node away
  else if (target.direction == "outbound"
     && (detector.CarQueue[target.systemId][0].systemId - target.systemId != 1)){
   distance = nodeArray[target.systemId].postDistance
       + nodeArray[detector.CarQueue[target.systemId][0].systemId].preDistance;
  }
  else{
    // do nothing use the default distance
  }

  // Calls the motionDetector's computeSpeed to calculate the object's speed
  // and the store the value in the target object
  target.speed = detector.computeSpeed(
                                    detector.CarQueue[target.systemId][0].currentTime,
                                    target.currentTime, distance);

  // Produses the speed history
  target.speedHistory = detector.CarQueue[target.systemId][0].speedHistory;
  target.speedHistory[target.systemId] = target.speed;

  // compute the time to be close to camera
  if (target.direction == "inbound"){
    target.timeToCamera = detector.computeTimeToCamera(
       nodeArray[target.systemId].distanceToCamera, target.speed);
  }

 // It calls the motionDetector's removeRedundantEntry which
  // checks and removes the old and redundant data from the arrays
```

```
    // in order to avoid future confusions
    detector.removeRedundantEntry(
        detector.CarQueue[target.systemId][0].seqNumber);
  }

  // The remaining prt of the method is responsible to store the above
  // prodused data in the proper node's data structure in order to infrom them
  // for the incoming object

  // if the target has been detected from node with system id 0
  if (target.systemId == nodeArray[0].systemId){

    if (target.direction == "unknown"){
      // If the node's data structure is full remove the first entry
      // to free space then it place the new entry
      if (detector.countersQueue[target.systemId + 1] ==
                                          detector.MAX_CAR_ARRAY_IDEX - 1) {
        // Removes the first element in the array to leave space
        // by calling the motionDetector rearrangeArray method. Rearange the
        // array from the first element "0" to the current array counter value.
        // It implements FIFO
        detector.rearrangeArray(0,target.systemId+1,
detector.countersQueue[target.systemId + 1],
                        detector.CarQueue);
        // Place the target object in the proper node array
        detector.CarQueue[target.systemId + 1]
                                [detector.countersQueue[target.systemId + 1]] = target;
      }
      // If the data structure is not full, it just places the new entry
      else {
        detector.CarQueue[target.systemId + 1]
                                [detector.countersQueue[target.systemId + 1]] = target;
        detector.countersQueue[target.systemId + 1]++;
```

```
    }

    // Increases system's reliability by placing the detection
    // exept the next node into one more node, two nodes away in the row.
    // It uses the the static topology that the application has based on
    // the system id. Thus, the current node with the static system id, and
    // the figure provided in the beggining of this method explain which
    // nodes the current node has to inform.
    // Then it implemtents the same steps as in the above set of selections
    if (detector.countersQueue[target.systemId + 2] ==
                                  detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 2,
                                  detector.countersQueue[target.systemId + 2],
                                  detector.CarQueue);
      detector.CarQueue[target.systemId + 2]
                          [detector.countersQueue[target.systemId + 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId + 2]
                          [detector.countersQueue[target.systemId + 2]] = target;
      detector.countersQueue[target.systemId + 2]++;
    }
  }
}

// The remainder of the method checks the system id of the node that
// returns the detection message, and the node's static topology
// and based on the stored direction informs the proper data structures
// following the steps that the above group of selection for the node with
// system id 0 implements.

// if the target has been detected from node with system id 1 to 6
if (target.systemId != nodeArray[0].systemId
```

```
  && target.systemId != nodeArray[7].systemId ){
// If the direction is inbound it informs only the nodes in that direction
if (target.direction == "inbound"){
  if (detector.countersQueue[target.systemId + 1] ==
                              detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                              detector.countersQueue[target.systemId + 1],
                              detector.CarQueue);
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
                    detector.countersQueue[target.systemId + 1]++;
  }
  // increases system reliability by placing the detection
  // exept the above next node into one more node if exist.
  // (e.g. it do not place the target object into the data
  // structure correspond to the system id 8 because it does not exist.
  if (target.systemId != nodeArray[6].systemId){

    if (detector.countersQueue[target.systemId + 2] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2],
                        detector.CarQueue);
      detector.CarQueue[target.systemId +
         2][detector.countersQueue[target.systemId + 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId +
         2][detector.countersQueue[target.systemId + 2]] = target;
```

```
      detector.countersQueue[target.systemId + 2]++;
    }
   }
  }
// If the direction is outbound it informs only the nodes in that direction
else if (target.direction == "outbound"){
  if (detector.countersQueue[target.systemId - 1] ==
                        detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 1,
                                detector.countersQueue[target.systemId - 1],
                        detector.CarQueue);
    detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
    detector.countersQueue[target.systemId - 1]++;
  }
  // increases system reliability by placing the detection
  // exept the above next node into one more node if exist.
  // (e.g. it do not place the target object into the data
  // structure correspond to the system id -1 because it does not exist.
  if (target.systemId != nodeArray[1].systemId){
    if (detector.countersQueue[target.systemId - 2] ==
                        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 2,
                        detector.countersQueue[target.systemId - 2],
                         detector.CarQueue);
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
    }
    else {
```

```
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }
  }
}

// If the direction is unkown it informs the nodes in both directions
else if (target.direction == "unknown"){
  if (detector.countersQueue[target.systemId + 1] ==
                         detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                        detector.countersQueue[target.systemId + 1],
                       detector.CarQueue);
  detector.CarQueue[target.systemId + 1]
                        [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                        [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

if (target.systemId != nodeArray[6].systemId){
  if (detector.countersQueue[target.systemId + 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2],
                        detector.CarQueue);
    detector.CarQueue[target.systemId +
        2][detector.countersQueue[target.systemId + 2]] = target;
  }
  else {
```

```
    detector.CarQueue[target.systemId +
        2][detector.countersQueue[target.systemId + 2]] = target;
    detector.countersQueue[target.systemId + 2]++;
   }
}

if (detector.countersQueue[target.systemId - 1] ==
                        detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1],
                      detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

if (target.systemId != nodeArray[1].systemId){
  if (detector.countersQueue[target.systemId - 2] ==
                        detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 2,
                        detector.countersQueue[target.systemId - 2],
                       detector.CarQueue);
    detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
    detector.countersQueue[target.systemId - 2]++;
```

```
      }
    }
  }
}

// if the target has been detected from node with system id 7
  if (target.systemId == nodeArray[7].systemId){
    if (target.direction == "unknown"){
      if (detector.countersQueue[target.systemId - 1] ==
                          detector.MAX_CAR_ARRAY_IDEX - 1) {
        detector.rearrangeArray(0, target.systemId - 1,
                          detector.countersQueue[target.systemId - 1],
                         detector.CarQueue);
        detector.CarQueue[target.systemId - 1]
                          [detector.countersQueue[target.systemId - 1]] = target;
      }
      else {
        detector.CarQueue[target.systemId - 1]
                          [detector.countersQueue[target.systemId - 1]] = target;
        detector.countersQueue[target.systemId - 1]++;
      }

      if (detector.countersQueue[target.systemId - 2] ==
                          detector.MAX_CAR_ARRAY_IDEX - 1) {
        detector.rearrangeArray(0, target.systemId - 2,
                          detector.countersQueue[target.systemId - 2],
                         detector.CarQueue);
        detector.CarQueue[target.systemId - 2]
                          [detector.countersQueue[target.systemId - 2]] = target;
      }
      else {
        detector.CarQueue[target.systemId - 2]
                          [detector.countersQueue[target.systemId - 2]] = target;
```

```
          detector.countersQueue[target.systemId - 2]++;
        }
       }
     }

  // call the motionDetector's sendCommand method to inform the camera
  detector.sendCommand(target.systemId, target.incomingObject,
                  target.direction, target.speed, target.timeToCamera,
                  target.speedHistory);
 }//End detectCar
          }




/**
 * <p>Title: TRoadScenario</p>
 * <p>Description: This class is responsible to implement the algorithmic
 *               process for the straight-road scenario by receiving the
 *               data from the motionDetector class.
 *               First based on the raw data and in the stored data it produces
 *               the direction of the object, and then the speed.
 *               Finally, it is responsible to inform the data structures of
 *               the neighbor nodes to continue tracking the object.</p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */

public class TRoadScenario {
  public TRoadScenario() {
```

```
}
 // Initialize a motionDetector object in order to be able to call the
 // proper methods
 private static motionDetector detector = new motionDetector();

 /**
  * Title: setDefaultDistances
  * Description: It takes the data related to the nodes physical characteristics
  *              and place them in a array of node objects.
  * @param initialNodeArray
  * @param userInputs
  * @return
  */
 public node[] setDefaultDistances(node[] initialNodeArray, int[][] userInputs) {
   //set the distances in the array that holds the data related to nodes
   initialNodeArray[0].setPreDistance(userInputs[0][1]);
   initialNodeArray[0].setPostDistance(userInputs[0][2]);
   initialNodeArray[1].setPreDistance(userInputs[1][1]);
   initialNodeArray[1].setPostDistance(userInputs[1][2]);
   initialNodeArray[2].setPreDistance(userInputs[2][1]);
   initialNodeArray[2].setPostDistance(userInputs[2][2]);
   initialNodeArray[3].setPreDistance(userInputs[3][1]);
   initialNodeArray[3].setPostDistance(userInputs[3][2]);
   initialNodeArray[4].setPreDistance(userInputs[4][1]);
   initialNodeArray[4].setPostDistance(userInputs[4][2]);
   initialNodeArray[5].setPreDistance(userInputs[5][1]);
   initialNodeArray[5].setPostDistance(userInputs[5][2]);
   initialNodeArray[6].setPreDistance(userInputs[6][1]);
   initialNodeArray[6].setPostDistance(userInputs[6][2]);
   initialNodeArray[7].setPreDistance(userInputs[7][1]);
   initialNodeArray[7].setPostDistance(userInputs[7][2]);

   initialNodeArray[0].setDistanceToCamera(
```

```
    setTRSNodeDistanceToCamera(initialNodeArray[0].systemId, initialNodeArray));
  initialNodeArray[1].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[1].systemId, initialNodeArray));
  initialNodeArray[2].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[2].systemId, initialNodeArray));
  initialNodeArray[3].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[3].systemId, initialNodeArray));
  initialNodeArray[4].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[4].systemId, initialNodeArray));
  initialNodeArray[5].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[5].systemId, initialNodeArray));
  initialNodeArray[6].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[6].systemId, initialNodeArray));
  initialNodeArray[7].setDistanceToCamera(
    setTRSNodeDistanceToCamera(initialNodeArray[7].systemId, initialNodeArray));
  return initialNodeArray;
}

/**
 * Title: setTRSNodeDistanceToCamera
 * Description: It calculates the distance from the node to the
 *              TSSRv3 camera-base station
 * @param systemId
 * @param initialNodeArray
 * @return
 */
public int setTRSNodeDistanceToCamera(int systemId, node[] initialNodeArray) {
  int distanceToCamera = 0;
  if (systemId == 0) {
    distanceToCamera = initialNodeArray[0].postDistance +
        initialNodeArray[1].postDistance +
        initialNodeArray[2].postDistance +
        initialNodeArray[4].postDistance +
```

```
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 1) {
  distanceToCamera = initialNodeArray[1].postDistance +
      initialNodeArray[2].postDistance +
      initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 2) {
  distanceToCamera = initialNodeArray[2].postDistance +
      initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 3) {
  distanceToCamera = initialNodeArray[3].postDistance +
      initialNodeArray[2].postDistance +
      initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 4) {
  distanceToCamera = initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
```

```
  else if (systemId == 5) {
    distanceToCamera = initialNodeArray[5].postDistance +
        initialNodeArray[6].postDistance +
        initialNodeArray[7].distanceLastNodeCamera;
  }
  else if (systemId == 6) {
    distanceToCamera = initialNodeArray[6].postDistance +
        initialNodeArray[7].distanceLastNodeCamera;
  }
  else if (systemId == 7) {
    distanceToCamera = initialNodeArray[7].distanceLastNodeCamera;
  }
  else {
    System.out.println(" Node Input Error ");
  }
  return distanceToCamera;
}

/**
 * Title: setDirection
 * Description: It implement the first part of the algorithm by producing the
 *              object's direction based on the node id that returns the
 *              detection and on the stored data in the node's data structure
 * @param sysId
 * @param quad
 * @return
 */
public String setDirection(int sysId, int quad) {

  String direction = "unknown";

  if (detector.CarQueue[sysId][0] == null) {
    direction = "unknown";
```

```
}

else if ( (sysId == 0 || sysId == 1 || sysId == 3)
        && sysId < detector.CarQueue[sysId][0].systemId) {
  if (detector.CarQueue[sysId][0].direction.startsWith("right", 8)) {
    direction = detector.CarQueue[sysId][0].direction;
  }
  direction = "right to left";
}
else if ( (sysId == 0 || sysId == 1 || sysId == 3)
        && sysId > detector.CarQueue[sysId][0].systemId) {
  if (detector.CarQueue[sysId][0].direction.startsWith("left", 8)) {
    direction = detector.CarQueue[sysId][0].direction;
  }
  direction = "left to right";
}

else if (sysId == 2) {
  if (detector.CarQueue[sysId][0].systemId == 2) {
    if (quad == 2) {
      if (detector.CarQueue[sysId][0].quad == 1) {
        direction = "inbound  right to left ";
      }
      else if (detector.CarQueue[sysId][0].quad == 2) {
      }
    }
    else if (quad == 1) {
      if (detector.CarQueue[sysId][0].quad == 1) {
      }
      else if (detector.CarQueue[sysId][0].quad == 2) {
        direction = "outbound left to right ";
      }
    }
```

```
  }
  else if (sysId > detector.CarQueue[sysId][0].systemId) {
    if (quad == 1) {
      direction = "left to right";
    }
    else if (quad == 2) {
      direction = "inbound  left to right";
    }
    else {
      direction = "left to right";
    }
  }
  else if (detector.CarQueue[sysId][0].systemId == 3) {
    direction = "right to left";
  }
  else if (sysId < detector.CarQueue[sysId][0].systemId
          && detector.CarQueue[sysId][0].systemId != 3) {
    direction = "outbound";
  }
}

else if (sysId == 4 || sysId == 5 || sysId == 6 || sysId == 7
        && sysId > detector.CarQueue[sysId][0].systemId) {
  if (detector.CarQueue[sysId][0].direction.startsWith("inbound")) {
    direction = detector.CarQueue[sysId][0].direction;
  }
  else {
    direction = "inbound";
  }
}
else if (sysId == 4 || sysId == 5 || sysId == 6 || sysId == 7
        && sysId < detector.CarQueue[sysId][0].systemId) {
  direction = "outbound";
```

```
 }
 return direction;
}

/**
 * Title: detect
 * Description: It implement the second part of the algorithm.
 *              By using the proper method's calls outputs the object's
 *              direction, and speed and stored them in the target object
 *              parameter.
 *              Its main purpose it to keep the the FIFO uptadated. Thus,
 *              by calling proper method's removes the old values. Aditioanlly
 *              by evaluating the objects direction it updates the proper node's
 *              data structures.
 *
 * The nodes topology that this scenario use based on the system id
 * is the following
 * The number represent the system id of the nodes
 *
 * -----------------1----------------------------------------3------------
 *
 *
 * ---0----------------------|               2---------------------------------
 *                           |               |
 *                           |               |
 *                           |               |
 *                          4                |
 *                           |               |
 *                           |               |
 *                           |               |
 *                           |               |
 *                           |              5
 *                           |               |
 *                           |               |
```

```
*                          |          |
*                        6            |
*                          |          |
*                          |          |
*                          |          |
*                          |          7
*
* @param node
*/
public void detect(motionDetector target, node[] nodeArray) {

 int distance = 0;

 // check if the stored data are too old
 if ( ( (long) target.currentTime - detector.oldTime) >=
    detector.timeThreshold) {
  // if the data are too old the resetCarArraysAndCounters() method
  // reset the counters and empty the arrays
  detector.resetCarArraysAndCounters();
 }

 // place the curent system time to the oldTime variable
 // in order to check the next incoming message
 detector.oldTime = (long) target.currentTime;
 //Test code displays the stored thresholds in the command line window
 System.out.println("PirThr" + detector.irThr + "MagThr" +
                detector.magneticThr);

 // Call the setDirection to calculate the direction of the object
 target.direction = setDirection(target.systemId, target.quad);

 // check if the node's data structure has stored target objects
 // If it has it use them  to calculate the object's speed
```

```
if (detector.CarQueue[target.systemId][0] != null) {
  //checks the direction of the object to place
  // the proper distance in the speed computation
  // if the direction is "unknown" it use s the default distance

  // place the proper distance if the stored value is from the neighbor node
  if (target.direction.startsWith("left") ||
      target.direction.startsWith("inbound")
      &&
      (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1)) {
    distance = nodeArray[target.systemId].preDistance;
  }
  else if (target.direction.startsWith("left") ||
          target.direction.startsWith("inbound")
          && (target.systemId == 4)
          && (detector.CarQueue[target.systemId][0].systemId == 2)) {
    distance = nodeArray[target.systemId].preDistance;
  }

  else if (target.direction.startsWith("left") ||
          target.direction.startsWith("inbound")
          && (target.systemId == 4)
          && (detector.CarQueue[target.systemId][0].systemId != 2)) {
    distance = nodeArray[target.systemId].preDistance
        + nodeArray[detector.CarQueue[target.systemId][0].systemId].postDistance;
  }
  // place the proper distance if the stored value is from the two node away
  else if (target.direction.startsWith("left") ||
          target.direction.startsWith("inbound") &&
          (target.systemId - detector.CarQueue[target.systemId][0].systemId !=  1)
          && (target.systemId != 4)) {
    distance = nodeArray[target.systemId].preDistance +
        nodeArray[detector.CarQueue[target.systemId][0].systemId].postDistance;
```

```
}

// place the proper distance if the stored value is from the neighbor node
else if (target.direction.startsWith("right") || target.direction.startsWith("outbound")
        && (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1)) {
  distance = nodeArray[target.systemId].postDistance;
}
else if (target.direction.startsWith("right") ||
        target.direction.startsWith("outbound")
        && (target.systemId == 2)
        && (detector.CarQueue[target.systemId][0].systemId == 4)) {
  distance = nodeArray[target.systemId].postDistance;
}
else if (target.direction.startsWith("right") || target.direction.startsWith("outbound")
        && (target.systemId == 2)
        && (detector.CarQueue[target.systemId][0].systemId != 4)) {
  distance = nodeArray[target.systemId].postDistance +
     nodeArray[detector.CarQueue[target.systemId][0].systemId].preDistance;
}
// place the proper distance if the stored value is from the two node away
else if (target.direction.startsWith("right") || target.direction.startsWith("outbound")
        && (target.systemId - detector.CarQueue[target.systemId][0].systemId != 1)) {
  distance = nodeArray[target.systemId].postDistance +
     nodeArray[detector.CarQueue[target.systemId][0].systemId].preDistance;
}

else {
  // do nothing use the default distance
}

// Calls the motionDetector's computeSpeed to calculate the object's speed
// and the store the value in the target object
target.speed = detector.computeSpeed(detector.CarQueue[target.systemId][0].
```

```
                currentTime, target.currentTime, distance);

    // Produses the speed history
    target.speedHistory = detector.CarQueue[target.systemId][0].speedHistory;
    target.speedHistory[target.systemId] = target.speed;

    // compute the time to be close to camera
    if (target.direction.startsWith("inbound")) {
      target.timeToCamera = detector.computeTimeToCamera(
          nodeArray[target.systemId].distanceToCamera, target.speed);
    }

    // It calls the motionDetector's removeRedundantEntry which
    // checks and removes the old and redundant data from the arrays
    // in order to avoid future confusions
    detector.removeRedundantEntry(
detector.CarQueue[target.systemId][0].seqNumber);
  }

  // The remaining prt of the method is responsible to store the above
  // produsd data in the proper node's data structure in order to infrom them
  // for the incoming object

  // if the target has been detected from node with system id 0
  if (target.systemId == nodeArray[0].systemId) {
    if (target.direction == "unknown") {
      // If the node's data structure is full remove the first entry
      // to free space then it place the new entry
      if (detector.countersQueue[target.systemId + 1] ==
          detector.MAX_CAR_ARRAY_IDEX - 1) {
        // Removes the first element in the array to leave space
        // by calling the motionDetector rearrangeArray method. Rearange the
        // array from the first element "0" to the current array counter value.
```

```
    // It implements FIFO
    detector.rearrangeArray(0, target.systemId + 1,
                        detector.countersQueue[target.systemId + 1], detector.CarQueue);
    // Place the target object in the proper node array
    detector.CarQueue[target.systemId + 1]
                [detector.countersQueue[target.systemId + 1]] = target;
  }
   // If the data structure is not full, it just places the new entry
  else {
    detector.CarQueue[target.systemId + 1]
                [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }
}
// Increases system's reliability by placing the detection
// exept the next node into one more node, two nodes away in the row.
// It uses the the static topology that the application has based on
// the system id. Thus, the current node with the static system id, and
// the figure provided in the beggining of this method explain which
// nodes the current node has to inform.
// Then it implemtents the same steps as in the above set of selections
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId +
      2][detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}
```

```
}

// The remainder of the method checks the system id of the node that
// returns the detection message, and the node's static topology
// and based on the stored direction informs the proper data structures
// following the steps that the above group of selection for the node with
// system id 0 implements.

// if the target has been detected from node with system id 2
if (target.systemId == nodeArray[2].systemId) {
  if (target.direction.startsWith("inbound")) {
    // inform node with system id 4
    if (detector.countersQueue[target.systemId + 2] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2], detector.CarQueue);
      detector.CarQueue[target.systemId + 2]
                [detector.countersQueue[target.systemId + 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId +
          2][detector.countersQueue[target.systemId + 2]] = target;
      detector.countersQueue[target.systemId + 2]++;
    }

    // it informa node with system id 5
    if (detector.countersQueue[target.systemId + 3] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 3,
                        detector.countersQueue[target.systemId + 3], detector.CarQueue);
      detector.CarQueue[target.systemId + 3]
                [detector.countersQueue[target.systemId + 3]] = target;
    }
```

```
  else {
    detector.CarQueue[target.systemId +
        3][detector.countersQueue[target.systemId + 3]] = target;
    detector.countersQueue[target.systemId + 3]++;
  }
}

if (target.direction == "outbound") {
  // inform node with system id 1
  if (detector.countersQueue[target.systemId - 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1], detector.CarQueue);
    detector.CarQueue[target.systemId - 1]
                [detector.countersQueue[target.systemId - 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 1]
                [detector.countersQueue[target.systemId - 1]] = target;
    detector.countersQueue[target.systemId - 1]++;
  }

  // informs node with system id 0
  if (detector.countersQueue[target.systemId - 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 2,
                        detector.countersQueue[target.systemId - 2], detector.CarQueue);
    detector.CarQueue[target.systemId -
        2][detector.countersQueue[target.systemId - 2]] = target;
  }
  // it place the new entry
  else {
    detector.CarQueue[target.systemId -
```

```
    2][detector.countersQueue[target.systemId - 2]] = target;
  detector.countersQueue[target.systemId - 2]++;
}

// Inform node with system id 2 itself, because node with system id 2
// uses two quarters
if (target.quad == 1 || target.quad == 2) {
  if (detector.countersQueue[target.systemId] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId,
                    detector.countersQueue[target.systemId], detector.CarQueue);
    detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
  }
  else {
    detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
    detector.countersQueue[target.systemId]++;
  }
}

// it infroms node with system id 3
else {
  if (detector.countersQueue[target.systemId + 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
```

```
      detector.countersQueue[target.systemId + 1]++;
    }
  }
}

if (target.direction.startsWith("left")) {
  // it informs the next node
  if (detector.countersQueue[target.systemId + 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                     [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }
}

if (target.direction.startsWith("right")) {
  // Informs node with system id 2 itself
  if (target.quad == 1 || target.quad == 2) {
    if (detector.countersQueue[target.systemId] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId,
                     detector.countersQueue[target.systemId], detector.CarQueue);
      detector.CarQueue[target.systemId]
                     [detector.countersQueue[target. systemId]] = target;
    }
    else {
      detector.CarQueue[target.systemId]
```

```
                    [detector.countersQueue[target. systemId]] = target;
    detector.countersQueue[target.systemId]++;
  }
}

// it infroms node with system id 4
else {
  if (detector.countersQueue[target.systemId + 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
    detector.countersQueue[target.systemId + 2]++;
  }
}

// inform node with system id 1
if (detector.countersQueue[target.systemId - 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                  detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                  [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
            [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
```

```
  }

  // informs node with system id 0
  if (detector.countersQueue[target.systemId - 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
   detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
  }
  else {
   detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
   detector.countersQueue[target.systemId - 2]++;
  }
 }

else if (target.direction == "unknown") {
  // Inform node with system id 2 itself
  if (target.quad == 1 || target.quad == 2) {
   if (detector.countersQueue[target.systemId] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId,
                     detector.countersQueue[target.systemId], detector.CarQueue);
    detector.CarQueue[target.systemId]
                     [detector.countersQueue[target. systemId]] = target;
   }
   else {
    detector.CarQueue[target.systemId]
                     [detector.countersQueue[target.systemId]] = target;
    detector.countersQueue[target.systemId]++;
   }
  }
```

```
// it place the entry in node with system id 3
if (detector.countersQueue[target.systemId + 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                     [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

// it place the entry in node with system id 1
if (detector.countersQueue[target.systemId - 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// it place the entry in node with system id 4
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
```

```
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs also node with system id 5
if (detector.countersQueue[target.systemId + 3] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 3,
                    detector.countersQueue[target.systemId + 3], detector.CarQueue);
  detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
}
else {
  detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
  detector.countersQueue[target.systemId + 3]++;
}

// informs node with system id 0
if (detector.countersQueue[target.systemId - 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
  detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
}
else {
```

```
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }
  }
}

// if the target has been detected from node is not with system id 0, 2, 3, 7
if (target.systemId != nodeArray[0].systemId
    && target.systemId != nodeArray[2].systemId
    && target.systemId != nodeArray[3].systemId
    && target.systemId != nodeArray[7].systemId) {

  if (target.direction.startsWith("inbound") ||
      target.direction.startsWith("left")) {
    // Inform's the next node
    if (detector.countersQueue[target.systemId + 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 1,
                        detector.countersQueue[target.systemId + 1], detector.CarQueue);
      detector.CarQueue[target.systemId + 1]
                        [detector.countersQueue[target.systemId + 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId + 1]
                        [detector.countersQueue[target.systemId + 1]] = target;
      detector.countersQueue[target.systemId + 1]++;
    }


    // The node with system id 1 in addition performs the following placements
    if (target.systemId == nodeArray[1].systemId) {
      // Informs node with system id 3
```

```
  if (detector.countersQueue[target.systemId + 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
   detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  }
  // it place the new entry
  else {
   detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
   detector.countersQueue[target.systemId + 2]++;
  }

  // informs node with system id 4
  if (detector.countersQueue[target.systemId + 3] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId + 3,
                    detector.countersQueue[target.systemId + 3], detector.CarQueue);
   detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
  }
  else {
   detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
   detector.countersQueue[target.systemId + 3]++;
  }
 }

// The nodes with system id 4, 5
else if (target.systemId == nodeArray[4].systemId ||
        target.systemId == nodeArray[5].systemId) {
  // Informs the second node in the row
```

```
    if (detector.countersQueue[target.systemId + 2] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2], detector.CarQueue);
      detector.CarQueue[target.systemId + 2]
                        [detector.countersQueue[target.systemId + 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId +
          2][detector.countersQueue[target.systemId + 2]] = target;
      detector.countersQueue[target.systemId + 2]++;
    }
  }
}

else if (target.direction.startsWith("outbound") ||
        target.direction.startsWith("right")) {
  // Informs the next node
  if (detector.countersQueue[target.systemId - 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1], detector.CarQueue);
    detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 1]
        [detector.countersQueue[target.systemId - 1]] = target;
    detector.countersQueue[target.systemId - 1]++;
  }

  // The node with system id 4 performs in addition the following
  if (target.systemId == nodeArray[4].systemId) {
```

```
  // Informs the node with system id 2
  if (detector.countersQueue[target.systemId - 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
   detector.CarQueue[target.systemId - 2]
      [detector.countersQueue[target.systemId - 2]] = target;
  }
  else {
   detector.CarQueue[target.systemId - 2]
      [detector.countersQueue[target.systemId - 2]] = target;
   detector.countersQueue[target.systemId - 2]++;
  }
}

// The node with system id 6 performs in addition the following
if (target.systemId == nodeArray[6].systemId) {
  // informs node with system id 4
  if (detector.countersQueue[target.systemId - 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
   detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
  }
  else {
   detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
   detector.countersQueue[target.systemId - 2]++;
  }
}

// The node with system id 4, 5 performs also the following
```

```
  if (target.systemId == nodeArray[5].systemId ||
     target.systemId == nodeArray[4].systemId) {
   // node with system id 5 informs node system id 2 and node with system id
   // 4 has already inform node with system id 2 and 3
   // and now informs node with system id 1
   if (detector.countersQueue[target.systemId - 3] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 3,
                    detector.countersQueue[target.systemId - 3], detector.CarQueue);
    detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 3]] = target;
   }
   else {
    detector.CarQueue[target.systemId - 3]
                    [detector.countersQueue[target.systemId - 3]] = target;
    detector.countersQueue[target.systemId - 3]++;
   }
  }
}

else if (target.direction == "unknown") {
  // Inform the next node
  if (detector.countersQueue[target.systemId + 1] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId + 1,
                  detector.countersQueue[target.systemId + 1], detector.CarQueue);
   detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
   detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
   detector.countersQueue[target.systemId + 1]++;
```

```
}

// The node with system id 1 it also performs the following
if (target.systemId == nodeArray[1].systemId) {
  // informs node with system id 3
  if (detector.countersQueue[target.systemId + 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2], detector.CarQueue);
    detector.CarQueue[target.systemId + 2]
                        [detector.countersQueue[target.systemId + 2]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 2]
                        [detector.countersQueue[target.systemId + 2]] = target;
    detector.countersQueue[target.systemId + 2]++;
  }
  // informs node with system id 4
  if (detector.countersQueue[target.systemId + 3] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 3,
                        detector.countersQueue[target.systemId + 3], detector.CarQueue);
    detector.CarQueue[target.systemId + 3]
                        [detector.countersQueue[target.systemId + 3]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 3]
                        [detector.countersQueue[target.systemId + 3]] = target;
    detector.countersQueue[target.systemId + 3]++;
  }
}

// The nodes with system id 4, 5 it also performs the following
```

```
else if (target.systemId == nodeArray[4].systemId ||
        target.systemId == nodeArray[5].systemId) {
  // exept the above next node into one more node two nodes in the row
  if (detector.countersQueue[target.systemId + 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
    detector.countersQueue[target.systemId + 2]++;
  }
}

// Inform the next node
if (detector.countersQueue[target.systemId - 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                   detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
     [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// The nodes with system id 4 it also performs the following
if (target.systemId == nodeArray[4].systemId) {
```

```
  // Informs the node with system id 2
  if (detector.countersQueue[target.systemId - 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
   detector.CarQueue[target.systemId - 2]
      [detector.countersQueue[target.systemId - 2]] = target;
  }
  else {
   detector.CarQueue[target.systemId - 2]
      [detector.countersQueue[target.systemId - 2]] = target;
   detector.countersQueue[target.systemId - 2]++;
  }
}

// The nodes with system id 6 it also performs the following
if (target.systemId == nodeArray[6].systemId) {
  // informs node with system id 4
  if (detector.countersQueue[target.systemId - 2] ==
     detector.MAX_CAR_ARRAY_IDEX - 1) {
   detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
   detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
  }
  else {
   detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
   detector.countersQueue[target.systemId - 2]++;
  }
}

// The nodes with system id 4, 5 it also performs the following
```

```
    if (target.systemId == nodeArray[5].systemId ||
        target.systemId == nodeArray[4].systemId) {
      // exept the above next node into one more node two nodes in the row
      // node with system id 5 informs node system id 2 and node with system id
      // 4 has already inform node with system id 2 and 3
      // and now informs node with system id 1
      if (detector.countersQueue[target.systemId - 3] ==
          detector.MAX_CAR_ARRAY_IDEX - 1) {
        detector.rearrangeArray(0, target.systemId - 3,
                        detector.countersQueue[target.systemId - 3], detector.CarQueue);
        detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 3]] = target;
      }
      else {
        detector.CarQueue[target.systemId - 3]
                        [detector.countersQueue[target.systemId - 3]] = target;
        detector.countersQueue[target.systemId - 3]++;
      }
    }
  }
}

// if the target has been detected from node with system id 7 or 3
if (target.systemId == nodeArray[7].systemId ||
    target.systemId == nodeArray[3].systemId) {
  if (target.direction == "unknown") {
    // Informs the next node
    if (detector.countersQueue[target.systemId - 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
```

```
      }
      else {
        detector.CarQueue[target.systemId - 1]
                          [detector.countersQueue[target.systemId - 1]] = target;
        detector.countersQueue[target.systemId - 1]++;
      }

      // informs nodes two node away
      if (detector.countersQueue[target.systemId - 2] ==
          detector.MAX_CAR_ARRAY_IDEX - 1) {
        detector.rearrangeArray(0, target.systemId - 2,
                          detector.countersQueue[target.systemId - 2], detector.CarQueue);
        detector.CarQueue[target.systemId - 2]
                          [detector.countersQueue[target.systemId - 2]] = target;
      }
      else {
        detector.CarQueue[target.systemId - 2]
                          [detector.countersQueue[target.systemId - 2]] = target;
        detector.countersQueue[target.systemId - 2]++;
      }
    }
  }

  // call the motionDetector's sendCommand method to inform the camera
  detector.sendCommand(target.systemId, target.incomingObject,
                       target.direction, target.speed, target.timeToCamera,
                       target.speedHistory);
 } //End detectCar
}
```

```
/**
 * <p>Title: crossroadsScenario</p>
 * <p>Description: This class is responsible to implement the algorithmic
 *              process for the straight-road scenario by receiving the
 *              data from the motionDetector class.
 *              First based on the raw data and in the stored data it produces
 *              the direction of the object, and then the speed.
 *              Finally, it is responsible to inform the data structures of
 *              the neighbor nodes to continue tracking the object.</p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Naval Postgraduate School, Monterey, CA</p>
 * @author Vlasios Salatas
 * @version 1.0
 */
public class crossroadsScenario {
  public crossroadsScenario() {
  }

  // Initialize a motionDetector object in order to be able to call the
  // proper methods
  private static motionDetector detector = new motionDetector();

  /**
   * Title: setDefaultDistances
   * Description: It takes the data related to the nodes physical characteristics
   *              and place them in a array of node objects.
   * @param initialNodeArray
   * @param userInputs
   * @return
   */
  public node[] setDefaultDistances(node[] initialNodeArray, int[][] userInputs){
```

```
//set the distances in the array that holds the data related to nodes
initialNodeArray[0].setPreDistance(userInputs[0][1]);
initialNodeArray[0].setPostDistance(userInputs[0][2]);
initialNodeArray[1].setPreDistance(userInputs[1][1]);
initialNodeArray[1].setPostDistance(userInputs[1][2]);
initialNodeArray[2].setPreDistance(userInputs[2][1]);
initialNodeArray[2].setPostDistance(userInputs[2][2]);
initialNodeArray[3].setPreDistance(userInputs[3][1]);
initialNodeArray[3].setPostDistance(userInputs[3][2]);
initialNodeArray[4].setPreDistance(userInputs[4][1]);
initialNodeArray[4].setPostDistance(userInputs[4][2]);
initialNodeArray[5].setPreDistance(userInputs[5][1]);
initialNodeArray[5].setPostDistance(userInputs[5][2]);
initialNodeArray[6].setPreDistance(userInputs[6][1]);
initialNodeArray[6].setPostDistance(userInputs[6][2]);
initialNodeArray[7].setPreDistance(userInputs[7][1]);
initialNodeArray[7].setPostDistance(userInputs[7][2]);

initialNodeArray[0].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[0].systemId, initialNodeArray));
initialNodeArray[1].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[1].systemId, initialNodeArray));
initialNodeArray[2].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[2].systemId, initialNodeArray));
initialNodeArray[3].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[3].systemId, initialNodeArray));
initialNodeArray[4].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[4].systemId, initialNodeArray));
initialNodeArray[5].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[5].systemId, initialNodeArray));
initialNodeArray[6].setDistanceToCamera(
   setCRNodeDistanceToCamera(initialNodeArray[6].systemId, initialNodeArray));
```

```
 initialNodeArray[7].setDistanceToCamera(
     setCRNodeDistanceToCamera(initialNodeArray[7].systemId, initialNodeArray));
 return initialNodeArray;
}

/**
 * Title: setTRSNodeDistanceToCamera
 * Description: It calculates the distance from the node to the
 *              TSSRv3 camera-base station
 * @param systemId
 * @param initialNodeArray
 * @return
 */
public int setCRNodeDistanceToCamera(int systemId, node[] initialNodeArray) {
 int distanceToCamera = 0;
 if (systemId == 0) {
   distanceToCamera = initialNodeArray[0].postDistance +
       initialNodeArray[1].postDistance +
       initialNodeArray[4].postDistance +
       initialNodeArray[5].postDistance +
       initialNodeArray[6].postDistance +
       initialNodeArray[7].distanceLastNodeCamera;
 }
 else if (systemId == 1) {
   distanceToCamera = initialNodeArray[1].postDistance +
       initialNodeArray[4].postDistance +
       initialNodeArray[5].postDistance +
       initialNodeArray[6].postDistance +
       initialNodeArray[7].distanceLastNodeCamera;
 }
 else if (systemId == 2) {
   distanceToCamera = initialNodeArray[2].postDistance +
       initialNodeArray[1].postDistance +
```

```
      initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 3) {
  distanceToCamera = initialNodeArray[3].postDistance +
      initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 4) {
  distanceToCamera = initialNodeArray[4].postDistance +
      initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 5) {
  distanceToCamera = initialNodeArray[5].postDistance +
      initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 6) {
  distanceToCamera = initialNodeArray[6].postDistance +
      initialNodeArray[7].distanceLastNodeCamera;
}
else if (systemId == 7) {
  distanceToCamera = initialNodeArray[7].distanceLastNodeCamera;
}
else {
  System.out.println(" Node Input Error ");
}
```

```
 return distanceToCamera;
}

/**
 * Title: setDirection
 * Description: It implement the first part of the algorithm by producing the
 *              object's direction based on the node id that returns the
 *              detection and on the stored data in the node's data structure
 * @param sysId
 * @param quad
 * @return
 */
public String setDirection(int sysId, int quad) {

  String direction = "unknown";
  // if the target is new it is characterized as unknown
  if (detector.CarQueue[sysId][0] == null) {
    direction = "unknown";
  }
  // the following set of selections related to the nodes around the crossing
  else if (sysId == 0 || sysId == 1 || sysId == 2 || sysId == 3 || sysId == 4) {

    switch (sysId) {
      //case for node with system id 0
      case 0: {
        // if the stored direction is unknown
        if (detector.CarQueue[sysId][0].direction == "unknown") {
          if (detector.CarQueue[sysId][0].systemId == 2) {
            direction = "north south to left";
          }
          else {
            direction = "right to left";
          }
```

```
  }
  // if the target came from the rest of the system and
  // it has already been detected
  else if (sysId < detector.CarQueue[sysId][0].systemId) {
    if ( (detector.CarQueue[sysId][0].direction.startsWith("right"))
        || (detector.CarQueue[sysId][0].direction.endsWith("right"))) {
      direction = detector.CarQueue[sysId][0].direction;
    }
    else if (! ( (detector.CarQueue[sysId][0].direction.startsWith("right")) ||
              (detector.CarQueue[sysId][0].direction.endsWith("right")))) {
      direction = detector.CarQueue[sysId][0].direction.concat(" left");
    }

    else {
      direction = "right to left";
    }
  }
}
break;

//case for node with system id 1
case 1: {
  //check if the stored data is from the same node with different quad
  if (sysId == detector.CarQueue[sysId][0].systemId) {
    if (quad == 1 && detector.CarQueue[sysId][0].quad == 2) {
      direction = "left to right north";
    }
    else if (quad == 2 && detector.CarQueue[sysId][0].quad == 1) {
      direction = "north to south left";
    }
  }
  // if the target comes from the node with system id 0
  else if (sysId > detector.CarQueue[sysId][0].systemId) {
```

```
  direction = "left to right";
}
// if the target comes from the rest of the nodes
else if (sysId < detector.CarQueue[sysId][0].systemId) {
  // if it comes from the node with system id 2
  if (detector.CarQueue[sysId][0].systemId == 2) {
    direction = "north to south";
  }
  else if (detector.CarQueue[sysId][0].systemId == 4
          || detector.CarQueue[sysId][0].systemId == 3) {
    if (quad == 1) {
      if (detector.CarQueue[sysId][0].quad == 1) {
        direction = "right to left north";
      }
      else if (detector.CarQueue[sysId][0].quad == 2) {
        direction = "outbound south to north";
      }
    }
    else if (quad == 2) {
      if (detector.CarQueue[sysId][0].quad == 1) {
        direction = "right to left";
      }
      else if (detector.CarQueue[sysId][0].quad == 2) {
        direction = "outbound south to north left";
      }
    }
  }
  // if the stored target has already valid direction
  else if (detector.CarQueue[sysId][0].direction.startsWith("right")
          || detector.CarQueue[sysId][0].direction.endsWith("right")) {
    direction = detector.CarQueue[sysId][0].direction;
  }
  // if the stored target has already valid direction
```

```
    else if (detector.CarQueue[sysId][0].direction.startsWith(
        "outbound")) {
      direction = detector.CarQueue[sysId][0].direction.concat(" left");
    }
    else if (detector.CarQueue[sysId][0].systemId == 3) {
      direction = "right to left";
    }
    else if (! ( (detector.CarQueue[sysId][0].systemId == 3)
              || (detector.CarQueue[sysId][0].systemId == 4))) {
      if (quad == 1) {
        direction = "outbound south to north";
      }
      else if (quad == 2) {
        direction = "outbound south to north left";
      }
    }
  }
} // end case 1
break;

case 2: {
  if (detector.CarQueue[sysId][0].direction == "unknown") {
    if (detector.CarQueue[sysId][0].systemId == 0) {
      direction = "left south to north";
    }
    else if (detector.CarQueue[sysId][0].systemId == 3) {
      direction = "right south to north";
    }
    else {
      direction = "south to north";
    }
  }
  else {
```

```
    if (sysId > detector.CarQueue[sysId][0].systemId) {
      if ( (detector.CarQueue[sysId][0].direction.startsWith("left"))
          || (detector.CarQueue[sysId][0].direction.startsWith("right"))) {
        direction = detector.CarQueue[sysId][0].direction.concat(
            " north");
      }
    }
    else if (sysId < detector.CarQueue[sysId][0].systemId) {
      if (detector.CarQueue[sysId][0].direction.startsWith("right")) {
        direction = detector.CarQueue[sysId][0].direction.concat(
            " north");
      }
      else if (detector.CarQueue[sysId][0].direction.startsWith(
          "outbound")) {
        direction = detector.CarQueue[sysId][0].direction;
      }
    }
  }
} // end case 2
break;

case 3: {
  if (detector.CarQueue[sysId][0].direction == "unknown") {
    if (sysId < detector.CarQueue[sysId][0].systemId
        && (! (detector.CarQueue[sysId][0].systemId == 4
            && detector.CarQueue[sysId][0].quad == 1))) {
      direction = "outbound left to right";
    }
    else {
      direction = "left to right";
    }
  }
  else {
```

```
    if ( (detector.CarQueue[sysId][0].direction.startsWith("left"))
        || (detector.CarQueue[sysId][0].direction.endsWith("left"))) {
      direction = detector.CarQueue[sysId][0].direction;
    }
    else {
      direction = detector.CarQueue[sysId][0].direction.concat(" left");
    }
  }
} // end case 3
break;

case 4: {
  //check if the stored data is from the same node with different quad
  if (sysId == detector.CarQueue[sysId][0].systemId) {
    if (quad == 2 && detector.CarQueue[sysId][0].quad == 1) {
      direction = "inbound right to left south";
    }
    else if (quad == 1 && detector.CarQueue[sysId][0].quad == 2) {
      direction = "outbound south to north right";
    }
  }
  else if (sysId > detector.CarQueue[sysId][0].systemId) {
    if (detector.CarQueue[sysId][0].direction == "unknown") {
      if (quad == 1) {
        if (detector.CarQueue[sysId][0].systemId == 3) {
          direction = "right to left";
        }
        else if (detector.CarQueue[sysId][0].systemId == 0) {
          direction = "left to right";
        }
        else if (detector.CarQueue[sysId][0].systemId == 1) {
          if (detector.CarQueue[sysId][0].quad == 2) {
            direction = "left to right";
```

```
      }
      else if (detector.CarQueue[sysId][0].quad == 1) {
        direction = "north to south right";
      }
    }
    else {
      direction = "north to south right";
    }
  }
  if (quad == 2) {
    if (detector.CarQueue[sysId][0].systemId == 3) {
      direction = "inbound right to left";
    }
    else if (detector.CarQueue[sysId][0].systemId == 0) {
      direction = "inbound left to right";
    }
    else if (detector.CarQueue[sysId][0].systemId == 1) {
      if (detector.CarQueue[sysId][0].quad == 2) {
        direction = "inbound left to right";
      }
      else if (detector.CarQueue[sysId][0].quad == 1) {
        direction = "inbound north to south";
      }
    }
    else {
      direction = "inbound north to south";
    }
  }
}

else {
  direction = "inbound ";
  direction.concat(detector.CarQueue[sysId][0].direction);
```

```
      }
     }
     if (sysId < detector.CarQueue[sysId][0].systemId) {
      if (detector.CarQueue[sysId][0].direction == "unknown") {
       if (quad == 1) {
        direction = "outbound south to north left";
       }
       else {
        direction = "outbound south to north";
       }
      }
      else {

       direction = detector.CarQueue[sysId][0].direction;
      }
     }
   } // end case 4
   break;
  } // end switch
}

else if (sysId == 5 || sysId == 6 || sysId == 7) {
  if (sysId > detector.CarQueue[sysId][0].systemId) {
   if (detector.CarQueue[sysId][0].direction == "unknown") {
    direction = "inbound north to south";
   }
   else if (detector.CarQueue[sysId][0].direction.startsWith("inbound")) {
    direction = detector.CarQueue[sysId][0].direction;
   }
   else if (!detector.CarQueue[sysId][0].direction.startsWith("inbound")) {
    direction = "inbound ";
    direction.concat(detector.CarQueue[sysId][0].direction);
   }
```

```
   }
   if (sysId < detector.CarQueue[sysId][0].systemId) {
    if (detector.CarQueue[sysId][0].direction == "unknown") {
      direction = "outbound south to north";
    }
    else {
      direction = detector.CarQueue[sysId][0].direction;
    }
   }
  }
  return direction;
}

/**
 * Title: detect
 * Description: It implement the second part of the algorithm.
 *              By using the proper method's calls outputs the object's
 *              direction, and speed and stored them in the target object
 *              parameter.
 *              Its main purpose it to keep the the FIFO uptadated. Thus,
 *              by calling proper method's removes the old values. Aditioanlly
 *              by evaluating the objects direction it updates the proper node's
 *              data structures.
 *
 * The nodes topology that this scenario use based on the system id
 * is the following
 * The number represent the system id of the nodes
 *
 *
 *                              |           |
 *                              |           |
 *                              |           2
 *                              |           |
```

```
*                              |             |
*                              |             |
*                              |             |
* -----------------------------1             |----------------3------------------
*
*
* -----------0-----------------|             4--------------------------------
*                              |             |
*                              |             |
*                              |             |
*                              |             |
*                              5             |
*                              |             |
*                              |             |
*                              |             |
*                              |             |
*                              |             6
*                              |             |
*                              |             |
*                              |             |
*                              |             |
*                              7             |
*
* @param node
*/
public void detect(motionDetector target, node[] nodeArray) {

 int distance = 0;

 // check if the stored data are too old
 if ( ( (long) target.currentTime - detector.oldTime) >=
    detector.timeThreshold) {
  // if the data are too old the resetCarArraysAndCounters() method
```

```
  // reset the counters and empty the arrays
  detector.resetCarArraysAndCounters();
}

// place the curent system time to the oldTime variable
// in order to check the next incoming message
detector.oldTime = (long) target.currentTime;
//Test code displays the stored thresholds in the command line window
System.out.println("PirThr" + detector.irThr + "MagThr" +
                detector.magneticThr);

// Call the setDirection to calculate the direction of the object
target.direction = setDirection(target.systemId, target.quad);

// check if the node's data structure has stored target objects
// If it has it use them  to calculate the object's speed
if (detector.CarQueue[target.systemId][0] != null) {

  //checks the direction of the object to place
  // the proper distance in the speed computation
  // if the direction is "unknown" it use s the default distance

  if (target.direction.startsWith("left")
      || target.direction.startsWith("inbound")
      || target.direction.startsWith("north")) {
    // if the stored data is from just the next node
    if ( (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1)
        || (detector.CarQueue[target.systemId][0].systemId - target.systemId == 1)) {
      distance = nodeArray[target.systemId].preDistance;
    }
    // if the stored data is from the node two steps away
    else if (! ( (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1) ||
            (detector.CarQueue[target.systemId][0].systemId - target.systemId == 1))) {
```

```
    distance = nodeArray[target.systemId].preDistance +
        nodeArray[detector.CarQueue[target.systemId][0].systemId].postDistance;
   }
}
else if (target.direction.startsWith("right") || target.direction.startsWith("outbound")
        || target.direction.startsWith("south")) {
  // if the stored data is from just the next node
  if ( (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1) ||
      (detector.CarQueue[target.systemId][0].systemId - target.systemId == 1)) {
    distance = nodeArray[target.systemId].postDistance;
  }
  // if the stored data is from the node two steps away
  else if (! ( (target.systemId - detector.CarQueue[target.systemId][0].systemId == 1) ||
            (detector.CarQueue[target.systemId][0].systemId - target.systemId == 1))) {
    distance = nodeArray[target.systemId].postDistance +
        nodeArray[detector.CarQueue[target.systemId][0].systemId].preDistance;
  }
}
else {
  // do nothing use the default distance
}

// Calls the motionDetector's computeSpeed to calculate the object's speed
// and the store the value in the target object
target.speed = detector.computeSpeed(detector.CarQueue[target.systemId][0].
                                currentTime, target.currentTime, distance);

// Produses the speed history
target.speedHistory = detector.CarQueue[target.systemId][0].speedHistory;
target.speedHistory[target.systemId] = target.speed;

// compute the time to be close to camera
if (target.direction.startsWith("inbound")) {
```

```
   target.timeToCamera = detector.computeTimeToCamera(
       nodeArray[target.systemId].distanceToCamera, target.speed);
  }

  // It calls the motionDetector's removeRedundantEntry which
  // checks and removes the old and redundant data from the arrays
  // in order to avoid future confusions

  detector.removeRedundantEntry(
      detector.CarQueue[target.systemId][0].seqNumber);
}

// The remaining prt of the method is responsible to store the above
// produsedd data in the proper node's data structure in order to infrom them
// for the incoming object

switch (target.systemId) {
  // if the target has been detected from node with system id 0
  case 0: {

    if (target.direction == "unknown") {
      // If the node's data structure is full remove the first entry
      // to free space then it place the new entry
      if (detector.countersQueue[target.systemId + 1] ==
          detector.MAX_CAR_ARRAY_IDEX - 1) {
        // Removes the first element in the array to leave space
        // by calling the motionDetector rearrangeArray method. Rearange the
        // array from the first element "0" to the current array counter value.
        // It implements FIFO
        detector.rearrangeArray(0, target.systemId + 1,
                          detector.countersQueue[target.systemId + 1], detector.CarQueue);
        // Place the target object in the proper node array
        detector.CarQueue[target.systemId + 1]
```

```
                    [detector.countersQueue[target.systemId + 1]] = target;
}
// If the data structure is not full, it just places the new entry
else {
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

// Increases system's reliability by placing the detection
// exept the next node into one more node, two nodes away in the row.
// It uses the the static topology that the application has based on
// the system id. Thus, the current node with the static system id, and
// the figure provided in the beggining of this method explain which
// nodes the current node has to inform.
// Then it implemtents the same steps as in the above set of selections

// informs node with system id 2
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
      [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs node with system id 4
if (detector.countersQueue[target.systemId + 4] ==
```

```
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 4,
                    detector.countersQueue[target.systemId + 4], detector.CarQueue);
    detector.CarQueue[target.systemId + 4]
                    [detector.countersQueue[target.systemId + 4]] = target;
   }
   else {
    detector.CarQueue[target.systemId + 4]
                    [detector.countersQueue[target.systemId + 4]] = target;
    detector.countersQueue[target.systemId + 4]++;
   }
  } // End target.direction == "unknown"
} // End case nodeArray[0].systemId
break;

// The remainder of the method checks the system id of the node that
// returns the detection message, and the node's static topology
// and based on the stored direction informs the proper data structures
// following the steps that the above group of selection for the node with
// system id 0 implements.

// if the target has been detected from node with system id 1
case 1: {
  if (target.direction == "unknown") {
   // Inform itself, because it uses two quarters
   if (detector.countersQueue[target.systemId] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId,
                    detector.countersQueue[target.systemId], detector.CarQueue);
    detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
   }
   else {
```

```
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
  detector.countersQueue[target.systemId]++;
}

// it informs the next node with system id 0
if (detector.countersQueue[target.systemId - 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// it informs the next node with system id 2
if (detector.countersQueue[target.systemId + 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}
```

```
// it informs the next node with system id 4
if (detector.countersQueue[target.systemId + 3] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 3,
                    detector.countersQueue[target.systemId + 3], detector.CarQueue);
  detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
}
else {
  detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
  detector.countersQueue[target.systemId + 3]++;
}

// informs node with system id 3
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs node with system id 5
if (detector.countersQueue[target.systemId + 4] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 4,
                    detector.countersQueue[target.systemId + 4], detector.CarQueue);
```

```
    detector.CarQueue[target.systemId + 4]
                    [detector.countersQueue[target.systemId + 4]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 4]
                    [detector.countersQueue[target.systemId + 4]] = target;
    detector.countersQueue[target.systemId + 4]++;
  }
} // End target.direction == "unknown"

// if the target direction starts with left
if (target.direction.startsWith("left")) {
  // if the target direction is left to right north
  if (target.direction.endsWith("north")) {
    // it informs the next node with system id 2
    if (detector.countersQueue[target.systemId + 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 1,
                      detector.countersQueue[target.systemId + 1], detector.CarQueue);
      detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
      detector.countersQueue[target.systemId + 1]++;
    }
  } // End direction.endsWith("north")

  // if the target direction is left to right
  else {
    // Inform itself, because it uses two quarters
    if (detector.countersQueue[target.systemId] ==
```

```
   detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId,
                    detector.countersQueue[target.systemId], detector.CarQueue);
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
}
else {
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target.systemId]] = target;
  detector.countersQueue[target.systemId]++;
}

// it informs the next node with system id 2
if (detector.countersQueue[target.systemId + 1] ==
   detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

// it informs the next node with system id 4
if (detector.countersQueue[target.systemId + 3] ==
   detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 3,
                    detector.countersQueue[target.systemId +3],
                    detector.CarQueue);
  detector.CarQueue[target.systemId + 3]
```

```
                    [detector.countersQueue[target.systemId + 3]] = target;
}
// it place the new entry
else {
  detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
  detector.countersQueue[target.systemId + 3]++;
}

// informs node with system id 3
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                      detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs node with system id 5
if (detector.countersQueue[target.systemId + 4] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 4,
                      detector.countersQueue[target.systemId + 4], detector.CarQueue);
  detector.CarQueue[target.systemId + 4]
                    [detector.countersQueue[target.systemId + 4]] = target;
}
else {
  detector.CarQueue[target.systemId + 4]
```

```
                    [detector.countersQueue[target.systemId + 4]] = target;
      detector.countersQueue[target.systemId + 4]++;
    }
  } // End else
} // End target.direction == "left"

// if the target direction starts with north
if (target.direction.startsWith("north")) {
  if (target.direction.endsWith("left")) {
    // it informs the next node with system id 0
    if (detector.countersQueue[target.systemId - 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                      detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId -1]
                    [detector.countersQueue[target.systemId - 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
      detector.countersQueue[target.systemId - 1]++;
    }
  } // End direction.endsWith("left")

  // if the target direction is north to south
  else {
    // Inform itself, because it uses two quarters
    if (detector.countersQueue[target.systemId] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId,
                      detector.countersQueue[target.systemId], detector.CarQueue);
      detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
```

```
}
// it place the new entry
else {
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
  detector.countersQueue[target.systemId]++;
}

// it informs the next node with system id 0
if (detector.countersQueue[target.systemId - 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                      detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
}
// it place the new entry
else {
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// it informs the next node with system id 4
if (detector.countersQueue[target.systemId + 3] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 3,
                      detector.countersQueue[target.systemId + 3], detector.CarQueue);
  detector.CarQueue[target.systemId + 3]
                    [detector.countersQueue[target.systemId + 3]] = target;
}
// it place the new entry
else {
```

```
  detector.CarQueue[target.systemId + 3]
                  [detector.countersQueue[target.systemId + 3]] = target;
  detector.countersQueue[target.systemId + 3]++;
}

// informs node with system id 3
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                  [detector.countersQueue[target.systemId + 2]] = target;
}
// it place the new entry
else {
  detector.CarQueue[target.systemId + 2]
                  [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs node with system id 5
if (detector.countersQueue[target.systemId + 4] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 4,
                    detector.countersQueue[target.systemId + 4], detector.CarQueue);
  detector.CarQueue[target.systemId + 4]
                  [detector.countersQueue[target.systemId + 4]] = target;
}
// it place the new entry
else {
  detector.CarQueue[target.systemId + 4]
                  [detector.countersQueue[target.systemId + 4]] = target;
  detector.countersQueue[target.systemId + 4]++;
```

```
    }
  } // End else
} // End direction.startsWith("north")

// if the target direction starts with right or outbound
if (target.direction.startsWith("right")
    || target.direction.startsWith("outbound")) {
  if (target.quad == 1) {
    // it informs the next node with system id 2
    if (detector.countersQueue[target.systemId + 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 1,
                        detector.countersQueue[target.systemId + 1], detector.CarQueue);
      detector.CarQueue[target.systemId + 1]
                        [detector.countersQueue[target.systemId + 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId + 1]
                        [detector.countersQueue[target.systemId + 1]] = target;
      detector.countersQueue[target.systemId + 1]++;
    }
  } // End target.quad == 1

  else if (target.quad == 2) {
    // it informs the next node with system id 0
    if (detector.countersQueue[target.systemId - 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
    }
    // it place the new entry
```

```
      else {
        detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
        detector.countersQueue[target.systemId - 1]++;
      }
    } // End target.quad == 2
  }
} // End case nodeArray[1].systemId
break;

// if the target has been detected from node with system id 2
case 2: {
  if (target.direction == "unknown") {
    // informs node with system id 1
    if (detector.countersQueue[target.systemId - 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
      detector.countersQueue[target.systemId - 1]++;
    }

    // informs node with system id 4
    if (detector.countersQueue[target.systemId + 2] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 2,
                        detector.countersQueue[target.systemId + 2], detector.CarQueue);
      detector.CarQueue[target.systemId + 2]
```

```
                    [detector.countersQueue[target.systemId + 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
      detector.countersQueue[target.systemId + 2]++;
    }

    // informs node with system id 0
    if (detector.countersQueue[target.systemId - 2] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
      detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }
  } // End target.direction == "unknown"
} // End case nodeArray[2].systemId
break;

// if the target has been detected from node with system id 3
case 3: {
  if (target.direction == "unknown") {
    // it informs the next node with system id 4
    if (detector.countersQueue[target.systemId + 1] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
```

```
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

// informs node with system id 5
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs node with system id 1
if (detector.countersQueue[target.systemId - 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
  detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
}
else {
  detector.CarQueue[target.systemId - 2]
```

```
                    [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }
  } // End target.direction == "unknown"
} // End case nodeArray[3].systemId
break;

// if the target has been detected from node with system id 4
case 4: {
  if (target.direction == "unknown") {
    // Inform itself, because it uses two quarters
    if (detector.countersQueue[target.systemId] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId,
                    detector.countersQueue[target.systemId], detector.CarQueue);
      detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
    }
    else {
      detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
      detector.countersQueue[target.systemId]++;
    }

    // it informs the next node with system id 3
    if (detector.countersQueue[target.systemId - 1] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId -1]
                    [detector.countersQueue[target.systemId - 1]] = target;
    }
    else {
```

```
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// it informs the next node with system id 5
if (detector.countersQueue[target.systemId + 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

// it informs the next node with system id 1
if (detector.countersQueue[target.systemId - 3] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 3,
                    detector.countersQueue[target.systemId - 3], detector.CarQueue);
  detector.CarQueue[target.systemId - 3]
                    [detector.countersQueue[target.systemId - 3]] = target;
}
else {
  detector.CarQueue[target.systemId - 3]
                    [detector.countersQueue[target.systemId - 3]] = target;
  detector.countersQueue[target.systemId - 3]++;
}
```

```
// informs node with system id 6
if (detector.countersQueue[target.systemId + 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}

// informs node with system id 2
if (detector.countersQueue[target.systemId - 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
  detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
}
else {
  detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
  detector.countersQueue[target.systemId - 2]++;
}

// informs node with system id 0
if (detector.countersQueue[target.systemId - 4] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 4,
                    detector.countersQueue[target.systemId - 4], detector.CarQueue);
```

```
    detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
    detector.countersQueue[target.systemId - 4]++;
  }
} // End target.direction == "unknown"

if (target.direction.startsWith("inbound")) {
  // it informs the next node with system id 5
  if (detector.countersQueue[target.systemId + 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }

  // informs node with system id 6
  if (detector.countersQueue[target.systemId + 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  }
```

```
  else {
    detector.CarQueue[target.systemId + 2]
                        [detector.countersQueue[target.systemId + 2]] = target;
    detector.countersQueue[target.systemId + 2]++;
  }
} // End target.direction == "inbound"

if (target.direction.startsWith("outbound")) {
  if (target.direction.endsWith("right")) {
    // it informs the next node with system id 3
    if (detector.countersQueue[target.systemId - 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                          detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
      detector.countersQueue[target.systemId - 1]++;
    }
  } // target.direction.endsWith("right")

  else {
    // Inform itself, because it uses two quarters
    if (detector.countersQueue[target.systemId] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId,
                          detector.countersQueue[target.systemId], detector.CarQueue);
      detector.CarQueue[target.systemId]
                        [detector.countersQueue[target. systemId]] = target;
    }
```

```
else {
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
  detector.countersQueue[target.systemId]++;
}

// it informs the next node with system id 3
if (detector.countersQueue[target.systemId - 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                      detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// it informs the next node with system id 1
if (detector.countersQueue[target.systemId - 3] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 3,
                      detector.countersQueue[target.systemId - 3], detector.CarQueue);
  detector.CarQueue[target.systemId - 3]
                    [detector.countersQueue[target.systemId - 3]] = target;
}
else {
  detector.CarQueue[target.systemId - 3]
                    [detector.countersQueue[target.systemId - 3]] = target;
  detector.countersQueue[target.systemId - 3]++;
}
```

```
    // informs node with system id 2
    if (detector.countersQueue[target.systemId - 2] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 2,
                        detector.countersQueue[target.systemId - 2], detector.CarQueue);
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }

    // informs node with system id 0
    if (detector.countersQueue[target.systemId - 4] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 4,
                        detector.countersQueue[target.systemId - 4], detector.CarQueue);
      detector.CarQueue[target.systemId - 4]
                        [detector.countersQueue[target.systemId - 4]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 4]
                        [detector.countersQueue[target.systemId - 4]] = target;
      detector.countersQueue[target.systemId - 4]++;
    }
  }
} // target.direction.startsWith("outbound")

if (target.direction.startsWith("right")) {
  // Inform itself, because it uses two quarters
```

```
if (detector.countersQueue[target.systemId] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId,
                    detector.countersQueue[target.systemId], detector.CarQueue);
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
}
else {
  detector.CarQueue[target.systemId]
                    [detector.countersQueue[target. systemId]] = target;
  detector.countersQueue[target.systemId]++;
}

// it informs the next node with system id 5
if (detector.countersQueue[target.systemId + 1] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
}
else {
  detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  detector.countersQueue[target.systemId + 1]++;
}

// it informs the next node with system id 1
if (detector.countersQueue[target.systemId - 3] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 3,
                    detector.countersQueue[target.systemId - 3], detector.CarQueue);
  detector.CarQueue[target.systemId - 3]
```

```
                    [detector.countersQueue[target.systemId - 3]] = target;
}
else {
  detector.CarQueue[target.systemId - 3]
                    [detector.countersQueue[target.systemId - 3]] = target;
  detector.countersQueue[target.systemId - 3]++;
}

// informs node with system id 2
if (detector.countersQueue[target.systemId - 2] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
  detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
}
else {
  detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
  detector.countersQueue[target.systemId - 2]++;
}

// informs node with system id 0
if (detector.countersQueue[target.systemId - 4] ==
    detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 4,
                    detector.countersQueue[target.systemId - 4], detector.CarQueue);
  detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
}
else {
  detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
```

```
      detector.countersQueue[target.systemId - 4]++;
    }
  } // target.direction.startsWith("right")

  if (target.direction.startsWith("left")) {
    // it informs the next node with system id 3
    if (detector.countersQueue[target.systemId - 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
      detector.countersQueue[target.systemId - 1]++;
    }
  } // End target.direction.startsWith("left")
} // End case nodeArray[4].systemId
break;

// if the target has been detected from node with system id 5
case 5: {
  if (target.direction.startsWith("inbound")) {
    // it informs the next node with system id 5
    if (detector.countersQueue[target.systemId + 1] ==
        detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
      detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    }
```

```
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }

  // informs node with system id 7
  if (detector.countersQueue[target.systemId + 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
    detector.countersQueue[target.systemId + 2]++;
  }
} // End target.direction == "inbound"

if (target.direction.startsWith("outbound")) {
  // it informs the next node with system id 4
  if (detector.countersQueue[target.systemId - 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
    detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
```

```
    detector.countersQueue[target.systemId - 1]++;
  }

  // informs node with system id 1
  if (detector.countersQueue[target.systemId - 4] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 4,
                    detector.countersQueue[target.systemId - 4], detector.CarQueue);
    detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
    detector.countersQueue[target.systemId - 4]++;
  }
} // End target.direction == "outbound"

if (target.direction == "unknown") {
  // it informs the next node with system id 5
  if (detector.countersQueue[target.systemId + 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }
```

```
// informs node with system id 7
if (detector.countersQueue[target.systemId + 2] ==
   detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId + 2,
                    detector.countersQueue[target.systemId + 2], detector.CarQueue);
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
}
else {
  detector.CarQueue[target.systemId + 2]
                    [detector.countersQueue[target.systemId + 2]] = target;
  detector.countersQueue[target.systemId + 2]++;
}
// it informs the next node with system id 4
if (detector.countersQueue[target.systemId - 1] ==
   detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
}
else {
  detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  detector.countersQueue[target.systemId - 1]++;
}

// informs node with system id 3
if (detector.countersQueue[target.systemId - 2] ==
   detector.MAX_CAR_ARRAY_IDEX - 1) {
  detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
  detector.CarQueue[target.systemId - 2]
```

```
                    [detector.countersQueue[target.systemId - 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }

    // informs node with system id 1
    if (detector.countersQueue[target.systemId - 4] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 4,
                    detector.countersQueue[target.systemId - 4], detector.CarQueue);
      detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 4]
                    [detector.countersQueue[target.systemId - 4]] = target;
      detector.countersQueue[target.systemId - 4]++;
    }
  } // End target.direction == "unknown"
} // End case nodeArray[5].systemId
break;

// if the target has been detected from node with system id 6
case 6: {
  if (target.direction.startsWith("inbound")) {
    // it informs the next node with system id 7
    if (detector.countersQueue[target.systemId + 1] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
```

```
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }
} // End target.direction == "inbound"

if (target.direction.startsWith("outbound")) {
  // it informs the next node with system id 5
  if (detector.countersQueue[target.systemId - 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
    detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
    detector.countersQueue[target.systemId - 1]++;
  }

  // informs node with system id 4
  if (detector.countersQueue[target.systemId - 2] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 2,
                    detector.countersQueue[target.systemId - 2], detector.CarQueue);
    detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
  }
```

```
  else {
    detector.CarQueue[target.systemId - 2]
                    [detector.countersQueue[target.systemId - 2]] = target;
    detector.countersQueue[target.systemId - 2]++;
  }
} // End target.direction == "outbound"

if (target.direction == "unknown") {
  // it informs the next node with system id 7
  if (detector.countersQueue[target.systemId + 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId + 1,
                    detector.countersQueue[target.systemId + 1], detector.CarQueue);
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId + 1]
                    [detector.countersQueue[target.systemId + 1]] = target;
    detector.countersQueue[target.systemId + 1]++;
  }

  // it informs the next node with system id 5
  if (detector.countersQueue[target.systemId - 1] ==
      detector.MAX_CAR_ARRAY_IDEX - 1) {
    detector.rearrangeArray(0, target.systemId - 1,
                    detector.countersQueue[target.systemId - 1], detector.CarQueue);
    detector.CarQueue[target.systemId - 1]
                    [detector.countersQueue[target.systemId - 1]] = target;
  }
  else {
    detector.CarQueue[target.systemId -
        1][detector.countersQueue[target.systemId - 1]] = target;
```

```
      detector.countersQueue[target.systemId - 1]++;
    }

    // informs node with system id 4
    if (detector.countersQueue[target.systemId - 2] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 2,
                        detector.countersQueue[target.systemId - 2], detector.CarQueue);
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }
  } // End target.direction == "unknown"
} // End case nodeArray[6].systemId
break;

// if the target has been detected from node with system id 7
case 7: {
  if (target.direction == "unknown") {
    // it informs the next node with system id 6
    if (detector.countersQueue[target.systemId - 1] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 1,
                        detector.countersQueue[target.systemId - 1], detector.CarQueue);
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
    }
    // it place the new entry
    else {
```

```
      detector.CarQueue[target.systemId - 1]
                        [detector.countersQueue[target.systemId - 1]] = target;
      detector.countersQueue[target.systemId - 1]++;
    }

    // informs node with system id 5
    if (detector.countersQueue[target.systemId - 2] ==
       detector.MAX_CAR_ARRAY_IDEX - 1) {
      detector.rearrangeArray(0, target.systemId - 2,
                        detector.countersQueue[target.systemId - 2], detector.CarQueue);
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
    }
    else {
      detector.CarQueue[target.systemId - 2]
                        [detector.countersQueue[target.systemId - 2]] = target;
      detector.countersQueue[target.systemId - 2]++;
    }
   } // End target.direction == "unknown
  } // End case nodeArray[7].systemId
  break;

  default:
   System.out.println("WARNING" + "\n" +
      "This version of the system is capable to handle up to " +
      "eight nodes!" + "\n" +
      "The data from the additional nodes are discarded");
} // End switch

// call the motionDetector's sendCommand method to inform the camera
detector.sendCommand(target.systemId, target.incomingObject,
                target.direction, target.speed, target.timeToCamera,
                target.speedHistory);
```

```
  } //End detectCar
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF REFERENCES


Al-Karaki, J., Kamal, A. (2005). *A Taxonomy of Routing Techniques in Wireless Sensor Networks.* In Ilayas, M., Mahgoub, I.. (Eds) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. Boca Raton.

Crossbow (2005). [http://www.xbow.com]. (last accessed 07/05)

Callaway H. (2004). Wireless Sensor Networks: Architectures and Protocols. Boca Raton.

Carle, J., Simplt-Ryl, D. (2004, February). Energy-Efficient Area Monitoring for Sensor Networks. *Computer, 37,* 40-46.

Culler, D., Hong, W. (Eds) (2004, June). Wireless Sensor Networks. *Communication of the ACM, 47,* 30-33.

Culler, D., Estin, D., Strivastava, M (2004, August) Overview of Sensor Networks. *Computer, 37, 41*-49.

Dixon B, Felts W.. Design and Implementation of a Networked Architecture Utilization Capabilities Discovery and Resource Aggregation. Monterey: Naval Postgraduate School, 2005.

Feibel W. (1995). The Encyclopedia of Networking. Alameda.

Haenggi, M. (2005). *Opportunities and Challenges in Wireless Sensor Networks*. In Ilayas, M., Mahgoub, I.. (Eds) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. Boca Raton.

Holger, K., Willig A. (2005). Protocols and Architectures for Wireless Sensor Networks. London.

IEEE. (2003). Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). New York

Karapetsas K.. Building a Simulation Toolkit for Wireless Mesh Clusters and Evaluating the Suitability of Different Families of Ad Hoc Protocols for the Tactical Network Topology. Naval Postgraduate School, Thesis, 2005.

Kurose J., Ross K.. Computer Networking: A Top-Down Approach Featuring the Internet. 2003.

MSP410 Datasheet [http://www.xbow.com]. (last accessed 07/05)

Ohrtman F., Roeder K.. Wi-Fi Handbook: Building 802.11b Wireless Networks. New York, 2003.

Perrig, A., Stankovic, J., Wagner, D. (2004, June). Security inWireless Sensor Networks. *Communication of the ACM, 47,* 53-57.

Peterson L., Davie B.. Computer Networks: A System Approach. San Francisco, 2003.

Stallings W.. Wireless Communication and Networks. New Jersey, 2002.

Slijepcevic, S., Wong, J., Potkonjak, M. (2005). *Security and Privacy Protection in Wireless Sensor Networks.* In Ilayas, M., Mahgoub, I.. (Eds) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. Boca Raton.

Wang, Q., Hassanein, H., Xu, K. (2005). *A Practical Perspective on Wireless Sensor Networks.* In Ilayas, M., Mahgoub, I.. (Eds) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. Boca Raton.

Wikipedia (2005). [http://en.wikipedia.org/wiki/Wireless_Mesh_Networks]. (last accessed 08/05).

Wireless sensor Networks: MTS/MDA Sensor and Data Acquisition Board User's Manual. San Jose: Crossbow, 2005.

Wireless sensor Networks: MPR/MIB User's Manual. San Jose: Crossbow, 2005.

Wireless sensor Networks: MOTE-VIEW 1.0 User's Manual. San Jose: Crossbow, 2005.

Wireless sensor Networks: MSP410 Mote Security User's Manual. San Jose: Crossbow, 2005.

Yarvis, M., Ye, W. (2005). *Tiered Architectures in Sensor Networks.* In Ilayas, M., Mahgoub, I.. (Eds) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. Boca Raton.

Zhao, F., Guibas, L. (2004). Wireless Sensor Networks: An Information Processing Approach. San Francisco.

THIS PAGE INTENTIONALLY LEFT BLANK

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Gurnider Singh
   Naval Postgraduate School
   Monterey, California

4. Arijit Das
   Naval Postgraduate School
   Monterey, California

5. Vlasios Salatas
   Hellenic Navy, Greece